KATHOLIEKE UNIVERSITEIT LEUVEN
FACULTEIT WETENSCHAPPEN
FACULTEIT TOEGEPASTE WETENSCHAPPEN
DEPARTEMENT COMPUTERWETENSCHAPPEN
Celestijnenlaan 200A — 3001 Leuven

FREENESS AND RELATED ANALYSES OF CONSTRAINT LOGIC PROGRAMS USING ABSTRACT INTERPRETATION

Jury :
Voorzitter Prof. Dr. ir. Y.D. Willems
Prof. Dr. ir. M. Bruynooghe, promotor
Prof. Dr. ir. M. Gobin, promotor
Prof. Dr. ir. R. Cools
Prof. Dr. D. De Schreye
Prof. M. Hermenegildo (U.P. Madrid, España)
Prof. B. Le Charlier (FUNDP, Namur)

Proefschrift voorgedragen tot
het behalen van het doctoraat
in de Informatica

door

Veroniek DUMORTIER

U.D.C. 681.3*D34,F31,I22,I23

Oktober 1994

Freeness and Related Analyses of Constraint Logic Programs Using Abstract Interpretation

Veroniek Dumortier

Department of Computer Science, K.U.Leuven

ABSTRACT

This thesis addresses the derivation of run-time properties of constraint logic programs. Constraint logic programming is more expressive than logic programming due to the combination with constraints. However, the increase in expressivity is often paid in terms of performance. For certain classes of queries, the full power of the constraint solvers is not needed. Run-time properties can then be used by an optimising compiler to generate specialised and efficient code.

Deriving run-time properties is formalised in terms of abstract interpretation. This is a method for interprocedural program analysis. We have used one of the existing analysis frameworks for logic programming that was already adapted towards constraint logic programming. The framework allows the developer of a particular program analysis to concentrate on the application-dependent parts and the associated safety conditions. An additional advantage of using an existing framework is the possibility to reuse implementations of the application-independent abstract interpretation procedure.

As an application of the framework, this thesis presents an analysis for determining possible constraint interaction. The analysis is named *freeness* analysis, as it infers which variables act as degrees of freedom with respect to the satisfiability of the constraints in which they occur. This information allows several optimisations such as constraint/goal reordering and parallelisation. The analysis focusses on constraint logic programming languages including the logic programming term domain and a numerical domain, although it can easily be extended to other constraint domains as well.

The efficiency of the kernel analysis can be improved in two ways : one approach is to retain only *minimal* information, the other consists of combining the freeness analysis with definiteness information. As the optimisations are orthogonal, they can be combined, yielding a practical and more complete analysis system. A number of extensions to the analysis (such as the treatment of non-normalised programs, the integration of type information, etc.) further enhance its power. The resulting system forms the basis for a large class of program optimisations.

Acknowledgements

This thesis has been accomplished with the help of a number of people.

First of all, I would like to thank my supervisor Prof. Maurice Bruynooghe for initiating me into the interesting domain of abstract interpretation and optimisation of (constraint) logic programs. His knowledge and wisdom have deepened my insight into this field. The research presented in this thesis has gained much from his valuable comments and suggestions.

Next, I would like to express my gratitude towards Prof. Marc Gobin who accepted to be my second supervisor. I thank both my supervisors and the other members of the thesis committee, Prof. Ronald Cools, Prof. Danny De Schreye, Prof. Manuel Hermenegildo, Prof. Baudouin Le Charlier and Prof. Yves Willems, for carefully reading this text and for their valuable feed-back.

Special thanks are due to Dr. Gerda Janssens for her daily support. Many inspiring discussions are at the basis of the ideas in this thesis. Her experience with abstract interpretation has led to a lot of useful suggestions.

I am largely indebted to Wim Simoens for his share in the prototype implementation and for reading first drafts of this text. He also helped me with issues related to text processing.

Thanks also go to Mike Codish for his comments on early versions of the analyses presented in this thesis. These led to a better formalisation and a clearer presentation of the analyses.

I would also like to thank Prof. Yves Willems for offering me the opportunity to join the logic programming group, and my colleagues in this group for the pleasant and stimulating atmosphere.

This research was funded in part by the Belgian government through the project RFO-AI-02 and by the EEC through the ESPRIT project #5246 PRINCE (PRolog INtegrated with Constraints and Environment for industrial and financial applications). I wish to thank the people of the PRINCE consortium for the helpful discussions and comments. I am especially grateful to the people of the CLIP team at the university of Madrid (Prof. Manuel Hermenegildo, Maria García de la Banda and Francisco Bueno). They provided the abstract interpretation system PLAI and the definiteness analysis for constraint logic programs. They offered immediate support when adaptations or extensions of the system were required. The actual communication of the definiteness and freeness analyses within PLAI has been established in collaboration between U.P.Madrid and K.U.Leuven.

Last but not least, I thank my parents for their continuous support.

Contents

Chapter 1

Introduction

A promising realisation in recent programming language design is the amalgamation of constraint programming and logic programming. Constraint programming [105, 106, 76, 102] is a powerful and natural programming paradigm, in which the objects of computation are not explicitly constructed but rather they are implicitly defined using constraints. This allows for very high-level programming, where the program directly refers to objects and relationships from the application domain and no special encoding (such as encoding into Herbrand terms in the case of logic programming) is required. Logic programming [101, 65, 74], on the other hand, is a convenient language for describing knowledge and rules for reasoning about this knowledge. It has a simple, declarative semantics. The combination, Constraint Logic Programming (CLP) [55, 54, 19, 18, 85], combines the best features of the two paradigms. It is an ideal tool for solving problems that require interactive mathematical modeling. Numerous applications have been developed in diverse areas such as electrical circuit analysis and synthesis [49], civil and mechanical engineering [103], options trading [67, 51] and financial planning [5], operations research (cutting stock [34], scheduling [12]), etc.

CLP can be considered as a generalisation of logic programming. The pattern-matching mechanism of unification, which is the core of logic programming, is replaced by the more general mechanism of constraint solving. Hence logic programming is just an instance of CLP involving one particular kind of constraints, namely term equations. In general however, CLP systems allow constraints over other domains and include domain-specific constraint solvers. For example, PrologIII [19, 100] supports constraint solving over the domain of rational numbers, Boolean values, tuples and infinite trees; CLP($\mathcal{R}$) [57] includes constraint solving over the domain of real numbers and the domain of terms (the Herbrand universe [74]).

Such CLP systems are very expressive and ideal for rapid prototyping. For some applications, the efficiency of a CLP program is comparable to or even better than what can be obtained by an equivalent program written in another (conventional) language, especially when the CLP program can take advantage of constraint-based pruning of the search space. However, often the expressive power is paid in terms of performance. General constraint solving should therefore be avoided if it is not really needed. For example, if all variables in a numerical constraint turn out to be instantiated by the time the constraint is encountered, a CLP compiler should transform it into a simple test; in this way, invocation of the expensive constraint solving algorithms (e.g. the simplex algorithm for inequalities) can

be avoided. Also higher-level optimisations such as constraint reordering may drastically improve efficiency [63, 85]. An optimising compiler [85] should incorporate a suite of optimisations in order to generate more optimal code. Determining the applicability of the different optimisations requires a large variety of dataflow information. This has motivated a growing interest in dataflow analysis of CLP programs. The work presented in this thesis was carried out in the context of the ESPRIT project PRINCE, which aims at developing a new CLP language and an optimising compiler that incorporates global analysis.

A technique that formalises global analysis is abstract interpretation [23]. The idea is to mimic the concrete interpretation of a program by using abstractions (approximations or descriptions) of the concrete data. Whereas considering all possible concrete evaluations of a program is in general intractable, using approximations one can obtain a dataflow analysis that terminates in finite time. Abstract interpretation has been an active area of research in the context of logic programming. A variety of frameworks and applications have been built, and some have shown great usefulness and practicality [72, 33, 112]. Integration into a compiler allows to generate high-quality code that is comparable to that obtained by a C compiler [107, 110]. Recently, abstract interpretation is also being applied in the context of CLP. The use of abstract interpretation for CLP was first discussed by Marriott and Sondergaard in [83]. Their framework encompasses a wide variety of programming languages, with a specific intention towards defining the (concrete and abstract) semantics of "logic-programming-like" languages. Afterwards, some other general frameworks for CLP (which includes LP) have been defined.

In the present work, we use an adaptation of one of the traditional frameworks for LP, more precisely the one of Bruynooghe [6]. Its adaptation for CLP was introduced by García de la Banda and Hermenegildo in [42, 43]. We mainly follow their description but put somewhat more emphasis on the *primitive* abstract operations [38, 36]. Also, we describe the overall abstract interpretation procedure in the same vein as the one in the original framework of Bruynooghe, except that termination is ensured in terms of a widening operator. The advantage of using the above framework is that we can reuse the efficient algorithm [94] that has been developed inspired by the framework. The extension towards CLP basically consists of replacing the LP notions *term domain, (abstract) substitution* and *(abstract) unification* by the CLP notions *constraint domain, (abstract) constraint* and *(abstract) conjunction* of a constraint to the set of constraints gathered so far. The safety conditions of the abstract operations on abstract constraints are reformulated accordingly. The result is a fully specified framework for the analysis of CLP programs. The application developer only has to design the application-dependent constraint descriptions and corresponding abstract operations. The basic issues in designing such a particular analysis of CLP programs are the safe and accurate abstraction of constraint entailment and the treatment of the interaction between different constraint domains. Especially the latter gives rise to additional complexity compared with the logic programming case.

Using the extended framework, this thesis presents an analysis for deriving information on possible constraint interaction. The analysis is termed *freeness* analysis, as it infers which variables act as degrees of freedom with respect to the satisfiability of the constraint set in which they occur. This can be viewed as a form of mode information. Integration of a limited form of type information is also briefly addressed. The information allows several optimisations such as constraint/goal reordering and parallelisation. The analysis focusses

on CLP languages including the logic programming term domain and a numerical domain, although it can be extended to include other constraint domains as well.

An optimised version of the analysis consists of keeping track only of *minimal* information, from which a safe approximation of the original abstraction can be reconstructed. The minimal freeness analysis can be considered as the practical realisation of the original freeness analysis.

The (minimal) freeness analysis is also combined with the definiteness analysis developed by García de la Banda and Hermenegildo [43, 41]. The latter was developed simultaneously within the PRINCE project and infers which variables are constrained to a unique value. The benefit of the combination is twofold: first, it allows to improve the efficiency of the freeness part, and secondly, it results in a full mode analysis system. The latter forms the basis of a larger suite of program optimisations, including for example constraint specialisation.

The different analyses have been implemented within the abstract interpretation system PLAI [94, 96]. This system was provided by the CLIP team at U.P.Madrid. They also generalised it to support the analysis of CLP programs and to allow the combination of their definiteness analysis with our freeness analyses that exploit definiteness information. The actual way of communication between the two analysers was studied in collaboration between U.P.Madrid and K.U.Leuven in the context of the PRINCE project.

A study of integrating the information derived by the analysers into a highly optimising CLP compiler is beyond the scope of this thesis. A discussion on compiler design and the incorporation of global analysers can be found in [85]. The main issues mentioned are (1) determining the applicability of each optimisation and (2) the non-trivial interaction between different optimisations (performing one optimisation in one part of the program may preclude performing another optimisation in another part). These topics require further research.

The text is organised as follows. In Chapters 2 and 3 the basic concepts of CLP and abstract interpretation are explained. Chapter 3 also formalises the extension of Bruynooghe's framework towards CLP. Chapter 4 discusses related work in the domains of abstract interpretation frameworks for CLP, optimisations of CLP programs and analyses that underlie these optimisations. Also related work on mode analysis in the context of logic programming is outlined. A situation of our work in the context of the related work is given. Chapter 5 contains the specification and safety proof of the kernel freeness analysis. It addresses only fundamental aspects and forms the basis for the more practical analyses of Chapters 6, 7 and 8. Two approaches are put forward to optimise the original freeness analysis. The first one consists of retaining only minimal information (Chapter 6). Using definiteness information, the second approach extracts the information that involves definite variables from the freeness abstraction and reduces it to the set of definite variables, without loss of information (Chapter 7). As the approaches are orthogonal, they can be combined together as described in Chapter 8. In Chapter 9, a number of extensions to the basic analyses of the previous chapters are presented. One involves the treatment of non-normalised programs. Another collection of extensions capture additional program properties, possibly changing the form of the abstraction. These include a more precise handling of disequations and inequalities and of passive constraints, the inference of definite and possible failure information and the addition of type information. A last extension

concerns the analysis of other constraint domains. Chapter 10 mentions the most important aspects of the implementation of the prototype analysers. In Chapter 11, the efficiency and accuracy of the prototypes are evaluated. It is also shown how the derived information can be used to optimise CLP programs. Finally, a conclusion is given in Chapter 12.

Chapter 2

Constraint Logic Programming

The first part of this chapter describes the basic concepts of Constraint Logic Programming (CLP) and introduces notational conventions. It also briefly addresses the issue of optimising CLP programs and systems, which motivates the development of global program analysers. In the second part, the emphasis is put on the subset of CLP languages that is dealt with in the rest of the thesis. We introduce the notion of solved form of a constraint. This form is needed to define the abstractions in Chapters 5, 6, 7 and 8. Finally, we define a simple typing of variables with respect to a constraint. We assume that the reader is familiar with the basic terminology of logic programming [74].

2.1 General concepts

Constraint Logic Programming (CLP) [55, 54, 19, 18, 109] combines the Logic Programming (LP) [101, 65, 74] and Constraint Programming [105, 106, 76, 102] paradigms. It allows the programmer to express a problem in a natural and declarative way and to reason with and about constraints. Programs are simple and concise and can be queried in many different ways. At the same time, constraints can be used in an active way (pruning the search space a priori) which may lead to efficient programs. The basic philosophy behind the introduction of CLP consists of replacing the pattern matching mechanism of unification – the core of LP – with the more general mechanism of constraint solving, thereby enhancing the expressive power. Computation is no longer restricted to symbolic computation over the Herbrand domain, but also includes (non-symbolic) computation over other domains such as the domain of real or rational numbers, the domain of Boolean values, etc. The domain of discourse D is just a parameter of the general CLP scheme, denoted CLP(D). Different instantiations of D give rise to different CLP languages, all of which share the same essential semantic properties as described in [54]. Some well-known instances of the scheme are the following:

- LP, i.e. CLP(H), where H denotes the Herbrand universe (the domain of LP terms) and where constraint solving consists of solving equations on terms;
- CLP($\mathcal{R}$) [57] which includes constraint solving over the Herbrand universe and over the domain of real numbers;

- PrologIII [19, 100] which allows constraint solving over the domain of rational trees, the domain of rational[1] numbers, the domain of Boolean values and the domain of tuples (also referred to as strings [55]);
- finite domain CLP languages such as CHIP [35], which allow constraint solving over a finite set of values (e.g. a finite subset of integers).

Example 2.1.1
To get an idea of the power and expressivity of CLP consider this small CLP($\mathcal{R}$) program:

$$celsius\_fahrenheit(C, F) :- F = 1.8 * C + 32.$$

This rule expresses the relation between the temperature in degrees Celsius (C) and Fahrenheit (F). The program can be used in many ways : if C is known then F can be computed and vice versa; if neither C nor F are known, the program still produces an interesting output, namely the constraint between C and F. The active use of constraints allows to prune the search space a priori, in contrast to using the constraints as tests if all involved variables have become known. E.g. if the constraints ($F = 1.8 * C + 32, F < 2.8 * C$) have been set up during execution, the value of C is immediately constrained to be larger than 32, thereby pruning off the search paths where C obtains a smaller value.

We now present some basic notions of CLP and introduce the notation that will be used throughout the rest of the thesis. We follow [54, 86, 102]. For more details we refer to [55]. Let *Var* be a denumerable set of variables, Σ a denumerable set of function symbols and $T(\Sigma, Var)$ the corresponding term algebra. Let $\Pi = \Pi_C \cup \Pi_P$ be a set of predicate symbols; Π_C contains the constraint symbols (including "=") which are pre-defined symbols over some computation domain and Π_P contains the programmer-defined predicate symbols, with $\Pi_C \cap \Pi_P = \emptyset$. A *primitive constraint* is constructed from a constraint symbol of Π_C and terms of $T(\Sigma, Var)$. E.g. the primitive constraints in LP are equations on terms. A primitive constraint is denoted by c (with or without subscript) in the sequel. A *constraint* is a (possibly empty) conjunction of primitive constraints, written as $c_1 \wedge \ldots \wedge c_n$. Constraints are denoted by C (with or without subscript) in the sequel. The fact that a primitive constraint c occurs in a constraint C is denoted as $c \in C$. An *atom* is constructed from a predicate symbol of Π_P and terms of $T(\Sigma, Var)$. A *literal* is an atom or a constraint. We let *Atom* denote the set of atoms, *Prim* the set of primitive constraints and *Cons* the set of constraints.
A constraint is said to be *satisfiable* iff it has at least one solution. A solution is an assignment σ of domain values to the variables in the constraint C, such that the domain theory (e.g. the theory of the domain of real numbers) implies $C\sigma$. We let *SCons* denote the set of satisfiable constraints.
Constraints are pre-ordered by logical implication, i.e. $C \leq C'$ iff $C \Rightarrow C'$; we also say that C *entails* C'. E.g. $(X = 3 \wedge Y = 4) \leq (X < Y)$. The (infinite) conjunction of all primitive constraints entailed by a constraint C is denoted C^*. Two constraints C_1 and C_2 are *equivalent* if $C_1 \leq C_2$ and $C_2 \leq C_1$; this implies that C_1 and C_2 have the same set of solutions. We let $\exists_W C$ be a non-deterministic function which returns a constraint logically equivalent to $\exists V_1 . \exists V_2 \ldots \exists V_n . C$ with the variable set $W = \{V_1, \ldots, V_n\}$. The

[1] The commercial system also allows computation over the domain of *real* numbers.

aim is to compute the simplest form with fewest quantifiers, but in general it is not possible to eliminate all uses of the existential quantifier. In the sequel, $\exists_W C$ is by definition also considered to be a constraint. We let $\overline{\exists}_W C$ be the constraint C restricted to the variables in W. That is $\overline{\exists}_W C$ is $\exists_{vars(C)\setminus W} C$ where the function *vars* takes a syntactic object and returns the set of (free) variables occurring in it.
A *constraint logic program* is a finite set of clauses of the form $h \leftarrow \mathcal{B}$, where h is an atom called the *head* and $\mathcal{B}$ is a sequence of the form $b_1, \ldots, b_n$ (the *body*) where each b_i is a literal. A *goal* is a (possibly empty) sequence of literals. Let *Prog* be the set of CLP programs.
A *renaming* is a bijective mapping from *Var* to *Var*. We let *Ren* be the set of renamings and naturally extend renamings to mappings between atoms, clauses and constraints. Syntactic objects s and s' are said to be *renamings* iff there is a $\rho \in Ren$ such that $\rho(s) = s'$.

The declarative semantics of CLP programs closely resembles the logical LP semantics. It interprets a clause $h \leftarrow b_1, \ldots, b_n$ as the logic formula $\forall X_1, \ldots, X_m . h \vee \neg b_1 \vee \ldots \vee \neg b_n$ where $X_1, \ldots, X_m$ are the variables in the clause; constraints are true if they are satisfiable with respect to the corresponding domain theory.

The operational semantics of a CLP program can be formulated in terms of its derivations which are reduction sequences of states. A *state* is a tuple consisting of the current goal and the current constraint (also called *current store*). In fact, the current store can be split up further into an active and a passive part (cf. [55]); the active part contains the constraints that are awake, whereas the passive part contains the constraints that are asleep/delayed and may be activated at a later execution step. For the moment, we focus on active constraints; passive constraints will be considered in Chapter 9. The basic reduction step in a CLP computation is similar to that of LP. However, instead of unification the more general notion of *constraint satisfiability* is applied. Let l be a literal (in fact, a sequence containing only the literal l), $\mathcal{G}$ a sequence of literals and C a constraint store; :: denotes concatenation of sequences. Assuming a left-to-right computation rule, a *reduction step* of state $s = \langle l :: \mathcal{G}, C \rangle$, *with* C *satisfiable*, for a program $P \in Prog$ returns a state s' such that

1. if $l \in Cons$ and $C' = (C \wedge l)$ is satisfiable, then $s' = \langle \mathcal{G}, C' \rangle$; otherwise, if C' is not satisfiable, $s' = \langle \mathcal{G}, false \rangle$;

2. if $l \in Atom$ and there exists a renaming $Cl : h \leftarrow \mathcal{B}$ of a clause in P, such that $vars(Cl) \cap vars(s) = \emptyset$, and $C' = (C \wedge (l = h))$ is satisfiable, then $s' = \langle \mathcal{B} :: \mathcal{G}, C' \rangle$. If there exists such a renaming Cl but C' is unsatisfiable, then $s' = \langle \mathcal{B} :: \mathcal{G}, false \rangle$. If there is no clause in P with the same predicate symbol p/k in its head as l, then $s' = \langle \mathcal{G}, false \rangle$.

Note that a state $\langle \ldots, false \rangle$ has no successor state (so unsatisfiability is not propagated). In the literature [55], a state of that form is sometimes referred to as the state *fail*; however, the format $\langle \ldots, false \rangle$ will lead to a more elegant formulation of the semantics constructions in subsequent chapters.
A *derivation* of a state s for a program P is a finite or infinite sequence of states $s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow \ldots$ where $s_0 = s$ and there is a reduction step from each s_i to s_{i+1}. The derivation is *successful* if the last state in the derivation has the empty goal and a constraint different from *false*. A constraint c is an *answer* to state s if there is a successful derivation from s to a state $\langle \epsilon, c \rangle$ where ϵ denotes the empty goal.

Checking for constraint satisfiability (at the same time performing constraint simplification) is done using domain-specific constraint solvers. Examples of such solvers are the well-known unification algorithm to solve syntactic equations over the Herbrand domain, the Gaussian elimination algorithm to deal with linear numerical equations, the simplex algorithm to solve linear inequalities, etc.

An important operational issue is performance. Increasing the efficiency of CLP execution can benefit from the results obtained in optimising the execution of LP (Prolog) programs. The most prominent techniques in that area are the specialisation of unification (Herbrand equation solving), based on mode analysis, and the reduction of choicepoint creation, based on the detection of deterministic predicates. Global program analysis provides the necessary information to guide source-to-source program transformation or to generate efficient code. In [110, 107], fairly efficient program analysers are integrated into the compilation of Prolog programs. The resulting code is comparable to that obtained from a C compiler and leads to impressive performance improvements. In the case of CLP, the opportunities for optimisation are even larger than in the case of Prolog. This is due to the fact that constraint solving is in general far more involved than unification. In the case of LP, there exist efficient algorithms [88] to find a most general unifier for a set of equations. However, for more general constraints such as numerical equations and inequalities, the algorithms for checking satisfiability of the constraint store each time a new constraint is added are more expensive. Applying these general solvers should therefore be avoided when possible. Experiments with current CLP systems show that a substantial speed-up is obtained by specialising constraints at compile-time, e.g. translating constraints into simple tests or assignments which avoids the invocation of the constraint solver [63, 56]. Even if the solver still has to be called, some special cases stand out [55]. Detection of these cases involves more or less complex global analysis. Besides these optimisations leading to special solver instructions, also higher-level optimisations are important to improve performance. These include program refinement [86] and a number of optimisations based on constraint or goal independence (constraint or goal reordering, parallelisation, and intelligent backtracking) [44]. All these optimisations are discussed in more detail in Chapter 4. They should be incorporated in a highly optimising compiler as the one designed in [85].

2.2 CLP(H,N)

In this thesis, we focus on CLP(H,N) programs which involve constraint solving over the Herbrand universe H and over an infinite domain of numbers N (more precisely the domain of rationals or reals, which forms a field for addition and multiplication). The language covers a relevant number of existing CLP languages : it covers a.o. LP, CLP($\mathcal{R}$) and part of PrologIII. It is not an oversimplification as the language already involves interactions between constraints of different constraint domains. Extension to other constraint domains is discussed in Chapter 9.

The set of function symbols Σ is divided into Σ_N and Σ_H (Σ_N and Σ_H overlap), containing respectively the *numerical* and *Herbrand* function symbols. The numerical function symbols will typically include: *numerical constants* being real or rational numbers – depending on the specific domain – and *numerical functors* such as $+, -, *$ and $/$. The Herbrand function

symbols include the constants (including numbers) and non-numerical function symbols. The corresponding term algebras are denoted respectively $T(\Sigma_N, Var)$ and $T(\Sigma_H, Var)$.

Two kinds of primitive constraints are distinguished : numerical constraints and unification constraints. A *primitive numerical constraint* is an equation, disequation or inequality between $T(\Sigma_N, Var)$ terms. In case it is linear, it is written in the form $a_1X_1 + \ldots + a_nX_n \diamond b$ where $\diamond \in \{=, \neq, >, \geq\}$, $a_1, \ldots, a_n, b$ are numbers with all $a_i \neq 0$ and the X_i are distinct variables; in case it is non-linear, it is written as $a_1t_1 + \ldots + a_nt_n \diamond b$ where $\diamond \in \{=, \neq, >, \geq\}$, $a_1, \ldots, a_n, b$ are numbers with all $a_i \neq 0$ and each t_i is of the form $X_{i1} \odot \ldots \odot X_{im}$ with $\odot \in \{*, /\}$ and the X_{ij} being variables $(1 \leq j \leq m)$. During execution, non-linear constraints are passive (i.e. delayed) until they become linear. A *primitive unification constraint* is an equation between $T(\Sigma_H, Var)$ terms. A *unification constraint* and *numerical constraint* are (possibly empty) conjunctions of respectively primitive unification constraints and primitive numerical constraints. A *mixed constraint* is a (possibly empty) conjunction of primitive unification and primitive numerical constraints. A unification constraint and numerical constraint can also be considered as special cases of a mixed constraint. Note that an equation of the form $X = f(Y + 1, 2 * Z - 3, T)$ cannot occur; it is *normalised* into $X = f(A, B, T) \wedge A - Y = 1 \wedge B - 2Z = 3$, consisting of a primitive unification and two primitive numerical constraints.

We let $num^*(C)$ and $unif^*(C)$ denote the respective conjunctions of all primitive numerical and all primitive unification constraints entailed by C; $C^* = num^*(C) \wedge unif^*(C)$. Note that primitive numerical constraints in C may entail a primitive unification constraint and vice versa.

Example 2.2.1
Let $C \equiv X = f(Y) \wedge X = f(Z)$. *Then*
$unif^*(C) \equiv X = f(Y) \wedge X = f(Z) \wedge Y = Z \wedge g(X) = g(f(Y)) \wedge \ldots$ *and*
$num^*(C) \equiv Y - Z = 0 \wedge 2Y - 2Z = 0 \wedge \ldots$
Let $C \equiv X + Y - Z = 3 \wedge Y = 3$. *Then*
$unif^*(C) \equiv X = Z \wedge g(X) = g(Z) \wedge \ldots$ *and*
$num^*(C) \equiv X + Y - Z = 3 \wedge Y = 3 \wedge X - Z = 0 \wedge 2X + 2Y - 2Z = 6 \wedge \ldots$

Note that $unif^*(C)$ and $num^*(C)$ are infinite conjunctions but they have a finite representation or so called *solved form*.

Each constraint C in CLP(H,N) can be transformed into an equivalent constraint called its *solved form*, denoted $sform(C)$. This solved form allows to decide whether two constraints are equivalent and to standardise output, making explicit information that was hidden in the original form of the constraint. The transformation of a constraint to solved form simplifies the constraint and at the same time checks it for satisfiability. The solved form of an unsatisfiable constraint such as $X + Y = 3 \wedge 2X + 2Y = 0$ or $a = b$ or $X = f(Y) \wedge X = g(Z)$ is *false*. A trivial satisfiable constraint such as $X + Y = X + Y$ or $a = a$ has as solved form *true*. Below, we define the solved form of a non-trivial constraint, distinguishing successively between a unification constraint, a numerical constraint and a mixed constraint.

A solved form of a satisfiable unification constraint C is a conjunction of primitive equations of the form

$$X_1 = t_1 \wedge \ldots \wedge X_n = t_n$$

with the X_i being distinct variables and $\{X_1, \ldots, X_n\} \cap vars(t_1, \ldots, t_n) = \emptyset$ (i.e. none of the X_i occurs in the right-hand sides of the equations). The variables in $vars(t_1, \ldots, t_n)$ are called the *parameters* of the solved form. Note that the solved form is unique up to the choice of parameters. It is also isomorphic to an idempotent substitution [68], more precisely to the most general unifier of the equations in C; hence, we do not distinguish them in the sequel. An algorithm to compute the solved form is described by Martelli and Montanari in [88].

Algorithm 2.2.1 (solved form of a unification constraint)
Let C be a unification constraint. The algorithm rewrites C to an equivalent constraint via a sequence of steps. At each step, one non-deterministically chooses an equation from the current constraint to which a numbered rule applies. The action taken is determined by the form of the equation :

1. $f(t_1, \ldots, t_m) = g(r_1, \ldots, r_n) \wedge C_{rest}$
   $\rightarrow$ *if $f \equiv g$ and $m \equiv n$ then transform the current constraint to $t_1 = r_1 \wedge \ldots \wedge t_n = r_n \wedge C_{rest}$; otherwise return false*

2. $X = X \wedge C_{rest}$
   $\rightarrow$ *transform to C_{rest}*

3. $t = X \wedge C_{rest}$ *where t is not a variable*
   $\rightarrow$ *transform to $X = t \wedge C_{rest}$*

4. $X = t \wedge C_{rest}$ *where $t \not\equiv X$ and X has another occurrence in C_{rest}*
   $\rightarrow$ *if $X \in vars(t)$ then return false; otherwise transform to $X = t \wedge C_{rest}[X/t]$, i.e. replace X by t in every equation of C_{rest}.*

The algorithm terminates when no step can be applied; $sform(C)$ is the constraint obtained at that point.

Example 2.2.2
Let $C \equiv X = f(Y) \wedge X = f(Z) \wedge a = T$. Then a solved form of C (with parameter Y) is $X = f(Y) \wedge Z = Y \wedge T = a$. An alternative solved form (with parameter Z) is $Y = Z \wedge X = f(Z) \wedge T = a$.

Concerning numerical constraints, the focus in the first part of the thesis (Chapters 5, 6, 7 and 8) is mainly on constraints that are conjunctions of linear equations. Their solved form is described below; this form has been presented in [111] and is also part of the solved form for linear constraints (including inequalities and disequations) described in [69]. The other numerical constraints are first treated in a safe but imprecise way, falling back on the treatment of equations. How to improve the treatment of numerical constraints in general is discussed in Chapter 9.
Now let C be a conjunction of linear equations. In that case, $num^*(C)$[2] is identical to

[2] at least, if $num^*(C)$ is also restricted to be a conjunction of equations, and so contains no inequalities or disequations

$lc(C)$, i.e. the conjunction of all linear combinations of the equations in C. Let $\bar{A}\bar{x} = \bar{b}$ represent the conjunction of linear equations in C where $\bar{A}$ is a mxn-matrix of numbers, $\bar{x}$ is a n-dimensional vector of variables and $\bar{b}$ is a m-dimensional vector of numbers; so each row in $\bar{A}\bar{x} = \bar{b}$ corresponds to one primitive equation in C. Computing the solved form of $\bar{A}\bar{x} = \bar{b}$ consists of computing the *reduced row-echelon form* $[\bar{A}' | \bar{b}']$ of the augmented matrix $[\bar{A} | \bar{b}]$ as mentioned in [111, 69] [3]. This form is characterised by (1) the first non-zero entry in a row (if any) is 1, (2) for any row i_0, let j_0 be the first column with a non-zero entry, then (2.a) for all $i > i_0$, $j \leq j_0$, $a'_{ij} = 0$, and (2.b) for all $i < i_0$, $a'_{ij_0} = 0$. The reduced row-echelon form is obtained by repeated application of one of three row operations : multiplication of a row by a non-zero number, addition of two rows or permutation of two rows.
In the solved form $\bar{A}'\bar{x} = \bar{b}'$, the leftmost variables of each row are solved in terms of the remaining variables which are called the *parameters* of the solved form. Based on the order of the variables in the vector $\bar{x}$ – i.e. based on the choice of parameters – a different solved form of C is obtained. Each solved form $\bar{A}'\bar{x} = \bar{b}'$ represents a constraint C' that is equivalent to the original constraint C. So the primitive equations in C' allow to produce every equation in $lc(C)$. An interesting property of each primitive constraint c in a solved form C' is that it cannot be obtained as a linear combination of some subset C_{sub} of C' such that[4] $vars(C_{sub}) \subset vars(c)$; intuitively, c is a kind of "minimal" element of $lc(C)$. If for some row in $[\bar{A}' | \bar{b}']$ all $a'_{ij} = 0$ and $b'_i \neq 0$, then the constraint C is unsatisfiable and its solved form is *false*.

Example 2.2.3
Let $C \equiv X - Y - Z = 0 \wedge T - Y - Z = 0 \wedge X - A = 3$; C can be represented in matrix notation as $\bar{A}\bar{x} = \bar{b}$:

$$\begin{bmatrix} 1 & 0 & 0 & -1 & -1 \\ 0 & 1 & 0 & -1 & -1 \\ 1 & 0 & -1 & 0 & 0 \end{bmatrix} \begin{bmatrix} X \\ T \\ A \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 3 \end{bmatrix}$$

Transformation to reduced row-echelon form yields :

$$\begin{bmatrix} 1 & 0 & 0 & -1 & -1 \\ 0 & 1 & 0 & -1 & -1 \\ 0 & 0 & 1 & -1 & -1 \end{bmatrix} \begin{bmatrix} X \\ T \\ A \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ -3 \end{bmatrix}$$

Here, X, T and A are solved in terms of the parameters Y and Z. The solved form of C with these parameters is $X - Y - Z = 0 \wedge T - Y - Z = 0 \wedge A - Y - Z = -3$.
Changing the order of the variables in x to Y, A, X, Z, T yields a solved form of C with

[3] The "reduced row-echelon form" in [111] corresponds to the "canonical solved form" in [69].
[4] $S_1 \subset S_2$ indicates that S_1 is a strict subset of S_2.

parameters Z and T :

$$\begin{bmatrix} 1 & 0 & 0 & 1 & -1 \\ 0 & 1 & 0 & 0 & -1 \\ 0 & 0 & 1 & 0 & -1 \end{bmatrix} \begin{bmatrix} Y \\ A \\ X \\ Z \\ T \end{bmatrix} = \begin{bmatrix} 0 \\ -3 \\ 0 \end{bmatrix}$$

The solved form is then $Y + Z - T = 0 \wedge A - T = -3 \wedge X - T = 0$.

For a mixed constraint C, the solved form consists of the solved forms of $unif^*(C)$ and $num^*(C)$, which are defined and computed by the following algorithm.

Algorithm 2.2.2 (solved form of a mixed constraint)
Let C be a mixed constraint, U the conjunction of primitive unification constraints in C and N the conjunction of primitive numerical constraints in C.

1. *Compute the solved form S_U of U using Algorithm 2.2.1. If $S_U \equiv false$, then halt with false (C is unsatisfiable); otherwise continue with step 2.*
2. *For every primitive constraint of the form $X = n$ or $X = Y$ in S_U where n is a number and X and Y are variables, add $X = n$ or $X - Y = 0$ to N. This yields N' (N augmented with constraints passed on from S_U).*
3. *Compute a solved form $S_{N'}$ of N' using the transformation to reduced row-echelon form. If $S_{N'} \equiv false$, then halt with false; otherwise continue with step 4.*
4. *For every primitive constraint of the form $aX = b$ or $X - Y = 0$ (with a and b being numbers ($a \neq 0$) and X and Y being variables) that can be obtained as a linear combination of the equations in $S_{N'}$[5], add $X = b/a$ or $X = Y$ to S_U; this yields an enlarged conjunction of primitive unification constraints, say U'.*
5. *Compute the solved form $S_{U'}$ of U' using Algorithm 2.2.1.*

If C turns out to be satisfiable, then $S_{U'}$ and $S_{N'}$ are defined to be the respective solved forms of $unif^*(C)$ and $num^*(C)$.

Passing information only once from the unification part to the numerical part (step 2) and back (step 4) is sufficient to obtain the solved forms of $unif^*(C)$ and $num^*(C)$. In other words, there is no need to repeat step 2 (and further) after step 5, since the only equations of the form $X = n$ or $X = Y$ that are in $S_{U'} \setminus S_U$ are already produced by the primitive numerical constraints in $S_{N'}$; the reasoning is as follows :

1. $X = n$ in $S_{U'} \setminus S_U$ is obtained in one of the following ways :
   (a) $X = n$ is added directly at step 4; then $X = n$ can be produced by $S_{N'}$.

[5] An alternative approach consists of computing *every* possible solved form of N' (corresponding with different parameter choices) and selecting from those solved forms the primitive constraints of the form $aX = b$ or $X - Y = 0$.

(b) $X = n$ arises from the combination of (1) $X = Y$ in S_U which implies that $X - Y = 0$ is put in N' at step 2 and (2) $Y = n$ that is produced by $S_{N'}$ at step 4. So both $X - Y = 0$ and $Y = n$ can be produced by $S_{N'}$ and therefore also $X = n$ can be produced by $S_{N'}$.

2. $X = Y$ in $S_{U'} \setminus S_U$ is obtained in one of the following ways :

(a) $X = Y$ is added directly at step 4; then $X - Y = 0$ can be produced by $S_{N'}$.

(b) $X = Y$ arises from the combination of (1) $X = Z$ in S_U which implies that $X - Z = 0$ is put in N' at step 2 and (2) $Z = Y$ that is put in U' at step 4 (i.e. $S_{N'}$ produces $Z - Y = 0$). So both $X - Z = 0$ and $Z - Y = 0$ can be produced by $S_{N'}$ and therefore also $X - Y = 0$ can be produced by $S_{N'}$.

(c) $X = Y$ arises from the combination of (1) $X = Z$ and $T = Y$ in S_U which implies that $X - Z = 0$ and $T - Y = 0$ are put in N' at step 2 and (2) $Z = T$ that is put in U' at step 4 (i.e. $S_{N'}$ produces $Z - T = 0$). Then $X - Z = 0$, $T - Y = 0$ and $Z - T = 0$ can be produced by $S_{N'}$ and therefore also $X - Y = 0$ can be produced by $S_{N'}$.

The termination of Algorithm 2.2.2 is based on the termination of the previous algorithms to transform a unification constraint and a numerical constraint to solved form (termination has been shown by the developers of these algorithms). It should be clear that all algorithms also preserve constraint equivalence.

Example 2.2.4
Let $C \equiv X = f(Z) \wedge X = f(Y) \wedge Y - Z - T + U = 0 \wedge W = X$. Then a solved form of C, with parameters Z and U, is given by

$$\begin{aligned} sform(unif^*(C)) &\equiv X = f(Z) \wedge Y = Z \wedge T = U \wedge W = f(Z) \text{ and} \\ sform(num^*(C)) &\equiv Y - Z = 0 \wedge T - U = 0. \end{aligned}$$

Finally, we introduce a simple typing of variables with respect to a constraint C. A variable $X \in Var$ is classified as either

- *Herbrand* if there exists a $X = t$ in $sform(unif^*(C))$ with $t \notin T(\Sigma_N, Var)$;
- *numerical* if X occurs in a primitive numerical constraint in C (i.e. a constraint of the form $aX + \ldots \diamond b$ with $\diamond \in \{=, \neq, >, \geq\}$), or if X is bound to a numerical variable Y via $X = Y$ or $Y = X$ in $sform(unif^*(C))$;
- *untyped* otherwise.

In the sequel, we use the type name *non-numerical* to refer to a variable that is either a Herbrand or an untyped variable.

Example 2.2.5
Let $C_1 \equiv X = Y \wedge Y = 3$; then X and Y are both numerical w.r.t. C_1. Let $C_2 \equiv X = Y \wedge T = f(X) \wedge 2A + B = 4$; then X and Y are untyped, T is Herbrand and A and B are numerical w.r.t. C_2. Let $C_3 \equiv T = f(X) \wedge X - Y = 0$; then T is Herbrand and X and Y are numerical w.r.t. C_3.

A constraint C is *well-typed* if Herbrand variables do not occur in (entailed) primitive numerical constraints (i.e. in $num^*(C)$). For example, $C \equiv X = Y \wedge Y = f(Z) \wedge X+3 = T$ is not well-typed since C entails $X = f(Z)$ and X also occurs in the primitive numerical constraint $X + 3 = T$. A variable X that is numerical, however, may occur in both $num^*(C)$ and $unif^*(C)$, e.g. consider X in $C \equiv Y = f(X) \wedge X - Z = 0$. Algorithm 2.2.2 can be extended with an extra step (step 6) to check for well-typedness :

6. If $X = t$ in S'_U with $t \notin T(\Sigma_N, Var)$ and X occurs in S'_N, then C is not well-typed and therefore halt with *false*.

So, constraints that are not well-typed are considered to be a subset of the set of non-satisfiable constraints.

Chapter 3

Abstract interpretation

The optimisations of CLP programs mentioned in Chapter 2 (e.g. compilation of constraints to assignments, etc.) are based on program information derived at compile-time. The technique of abstract interpretation formalises the compile-time analysis. We first set forth the basic concepts of abstract interpretation. Then we sketch one framework in particular, as it is used in the remainder of the thesis. Although originally designed for LP, this framework has easily been adapted towards CLP [43, 41]. For a more detailed introduction to abstract interpretation of declarative languages we refer to [1, 33].

3.1 Basic concepts

Static program analysis is a technique to derive properties of the run-time behaviour of a program. The derived information can be used to optimise the code generated by a compiler or to guide source level program transformation and program development tools. Program analysis can be viewed as executing a program over a domain of *data descriptions* or *abstract data*, instead of using the concrete data; this led to the name *abstract interpretation*. The domain of data descriptions is called the *abstract domain*. The descriptions approximate the concrete data, reflecting some specific properties of those. The operations on the concrete domain are replaced with *abstract operations* on the abstract domain. These abstract operations must mimic the concrete execution in order for the analysis to be *safe* (also called *sound*). The resulting analysis derives for each control point in the program a finite description of the set of data that may occur when execution passes that point.

Example 3.1.1
A well-known example from mathematics is the sign analysis. The interest is in deriving the sign of the result when performing some numerical operation like multiplication of two numbers. Assume that the concrete domain, i.e. the set of concrete data, is the set of integer numbers. The abstract domain is the set of possible signs, i.e. $\{+,-,0\}$*; "+" describes the set of strictly positive numbers, "−" the set of strictly negative numbers and "0" the number zero. Abstract multiplication* $*^a$ *is defined by the following table :*

$*^a$	−	0	+
−	+	0	−
0	0	0	0
+	−	0	+

This abstract multiplication over the sign domain is the counterpart of the concrete multiplication over the domain of integers.

A general framework for abstract interpretation has first been developed by Patrick and Radhia Cousot [23]. It presents a semantic approach to data-flow analysis where the analysis is viewed as a non-standard abstract semantics which approximates the concrete semantics. In fact, the conventional approach is to lift the concrete semantics (such as e.g. the operational semantics in Chapter 2), which describes subsequent execution states of a program, to a powerdomain of states, resulting in a so called *collecting semantics* [78]. The idea is that during program execution control may return to a program point many times, so a point naturally corresponds to a collection of states rather than a single state. All successor states of a state in the concrete semantics are bundled into a collecting state. The abstract description at some program point then describes this collecting state. Based on that pioneer work, a variety of frameworks (and applications) have been developed in the context of LP (e.g. [6, 25, 33, 31, 82, 70, 89, 97]). A framework is a generic construction that provides a basis for the computation of safe abstractions. It includes general conditions that ensure the safety and termination of the analysis. A particular analysis is obtained by plugging in the application dependent domains and operations (concrete or abstract ones). The use of abstract interpretation for CLP has been addressed in [83, 17, 46, 11]. Further discussion of the frameworks for abstract interpretation of CLP is deferred to Chapter 4.

We now sketch the basic notions of abstract interpretation based on the traditional formulation of Cousot&Cousot [23]. Let $(Dom^c, \leq^c)$ be the concrete domain and $(Dom^a, \leq^a)$ be the domain of descriptions. In the context of LP, the concrete domain is typically the powerset of the set of substitutions (with $\leq^c$ defined as $\subseteq$); the abstract domain is the set of so called abstract substitutions, which describe a set of concrete substitutions. For CLP, the concrete domain is typically the powerset of the set of constraints (again with $\leq^c$ defined as $\subseteq$); the abstract domain is the set of abstract constraints, where each abstract constraint corresponds to a set of concrete ones. The relation between concrete and abstract domain is formalised by means of an *abstraction function* $\alpha : Dom^c \rightarrow Dom^a$ and a *concretisation function* $\gamma : Dom^a \rightarrow Dom^c$. Given a set of concrete data, the abstraction function yields the best description of these data. Conversely, the concretisation function yields the set of concrete data that are described by some element of the abstract domain. Suppose that the standard interpretation of a program is defined in terms of a semantic function F^c, which has an abstract counterpart F^a. The fixpoint of F^c and F^a corresponds to respectively the standard and abstract semantics of the program. For logic programs, unification is the central operation within F^c. The counterpart in CLP is the conjunction of a constraint to the current store. The traditional abstract interpretation scheme [23] requires that the following conditions are satisfied:

1. $(Dom^c, \leq^c)$ and $(Dom^a, \leq^a)$ are complete lattices, i.e. sets with a partial order such that each subset has a least upper bound and a greatest lower bound,

2. $\alpha : Dom^c \rightarrow Dom^a$ and $\gamma : Dom^a \rightarrow Dom^c$ are monotonic, i.e.

   - $x \leq^c y \Rightarrow \alpha(x) \leq^a \alpha(y)$, for all $x, y \in Dom^c$,
   - $x \leq^a y \Rightarrow \gamma(x) \leq^c \gamma(y)$, for all $x, y \in Dom^a$,

3. $\alpha(\gamma(x)) \leq^a x$ for all $x \in Dom^a$,

4. $x \leq^c \gamma(\alpha(x))$ for all $x \in Dom^c$,

5. $F^c : Dom^c \rightarrow Dom^c$ and $F^a : Dom^a \rightarrow Dom^a$ are monotonic,

6. $F^c(\gamma(x)) \leq^c \gamma(F^a(x))$ for all $x \in Dom^a$.

Conditions 1, 2, 3 and 4 imply that $(Dom^c, \leq^c) \underset{\alpha}{\overset{\gamma}{\rightleftarrows}} (Dom^a, \leq^a)$ is a *Galois connection*[1]. Note that condition 1 can be weakened to allow other algebraic structures as described in [26]. If condition 3 is strengthened by replacing $\leq^a$ with $=^a$, then one obtains a so called *Galois insertion*. Condition 5 ensures that the iterates in the fixpoint computation are in increasing order with respect to $\leq^c$ in the concrete case or $\leq^a$ in the abstract case. The last condition is the *safety* requirement and ensures that the abstract operations mimic the concrete ones such that the abstraction correctly describes the program behaviour.
In the context of a Galois connection, α and γ are adjoint, i.e. one function uniquely determines the other (cf. [25]). Consequently, only one of the two needs to be defined (the practical impact is that one may be easier to define than the other).

Two basic classes of abstract interpretation of (C)LP programs can be distinguished : top-down and bottom-up analyses. The *top-down* analyses are based on the standard SLD-semantics. They are query-driven, i.e. a *query specification* is provided that characterises how the program is used. Based on this specification, information is derived about the states in which clauses and calls can be reached. Both call and success information, i.e. information valid just before and just after a predicate call, can be inferred. The *bottom-up* analyses [14, 80] are based on the T_p-semantics [74]. Because this semantics does not reflect the operational behaviour of a program, it does not readily provide information about call patterns of goals. Furthermore, bottom-up analyses tend to derive a lot of redundant facts that are not actually needed to answer a specific query. Recently, a hybrid approach has been proposed : the program and query are first transformed using the magic templates method and the resulting program is then analysed bottom-up [98, 14].

3.2 Abstract interpretation framework for (C)LP

In this section, we explain the abstract interpretation framework of Bruynooghe [6] that forms the basis of our analysis. It is a top-down analysis method that allows to derive call and success descriptions of literals, starting off from an initial query specification that is supplied by the user. Developing a specific analysis within the framework consists of designing the application dependent components, i.e. the abstract domain and the abstract operations that are used within the abstract interpretation procedure, which is itself application independent. The abstract operations should satisfy certain safety conditions, which imply the safety of the entire analysis.
Although originally designed for LP, the framework can easily be adapted for CLP, as described by García de la Banda and Hermenegildo in [43, 41]. Since the control flow of CLP execution is identical to that of LP execution (i.e. based on a left-to-right depth-first

[1] In order to have a Galois connection, it suffices that Dom^c and Dom^a are sets with a partial order and not necessarily complete lattices as stated in condition 1 (cf. [26]).

computation rule[2]), the same abstract interpretation procedure can be used. The adaptation for CLP consists of replacing the notions of *Herbrand domain, (abstract) substitution* and *(abstract) unification* with *constraint domain, (abstract) constraint* and *(abstract) conjunction of a constraint to the store*. The safety conditions are adjusted accordingly. In this section we slightly reformulate the description of [43]. More emphasis is put on the distinction between *primitive* and other abstract operations. The framework is also modified in another aspect (not specific to CLP) : termination is ensured in terms of the more standard notion of a widening operator [27]. At the end of the section, we present the overall abstract interpretation procedure, formulated in the same vein as the one in the original framework of Bruynooghe.
For the original description of the framework tuned towards LP we refer to [6]. Below we elaborate its CLP version.

3.2.1 General characteristics

In the framework, data-descriptions are derived at a fixed number of *program points*: at the beginning of each clause, in between two body literals and at the end of each clause. In the context of CLP, the description in front of a literal is called its *abstract call constraint*. The information to the right of a literal is called its *abstract success constraint*.

The framework is organised around an AND-OR-graph. Such a graph covers a set of (possibly infinite) AND-OR-trees. Each of these AND-OR-trees represents a concrete execution of the program for a query that belongs to the given query specification. The organisation of the computation in an AND-OR tree is based on bundling together states in the standard concrete semantics into collecting states (giving rise to the collecting semantics [78] as mentioned in the previous section). An OR-node corresponds to a call and its children represent the different clauses that match with the call. An AND-node represents a clause head and has one offspring for each of the goals in the clause body. Several concrete OR-nodes along the same branch of an AND-OR-tree that correspond with recursive calls having the same abstract call-pattern collapse into a single OR-node in the AND-OR-graph. The graph is adorned with abstract information, more precisely information is added to the left and right of each OR-node (corresponding to the program points in the program). The information to the left of the root node is the query specification that is provided by the user and that describes a set of concrete queries. All abstract descriptions are restricted to the variables in the clause or query that they adorn. This set of variables is called the *domain* of the description. By confining attention to these variables, the analysis can recognise the similarity between different invocations of a clause that differ only in the context of their use. This property is essential for the treatment of recursive calls. It also allows to prevent repeated subcomputations (i.e. repeated computation of the same subgraph in different branches of the AND-OR-graph).

The AND-OR-graph provides the information that is needed for program manipulation (transformation, compilation,...). In general, the AND-OR-graph may contain several instances of a clause, corresponding to different call-patterns. This allows to derive different specialised versions of a procedure. Whether specialisation is actually performed or not, is left to the program manipulation tool. For example, a compiler has the choice between

[2]This is not really true when passive constraints are involved. It will be further discussed in Chapter 9.

generating code for different versions of a procedure (taking care of calling the appropriate version at each invocation) or generating general code that encompasses each version.

The construction of the AND-OR-graph is based on the control flow in (C)LP. The standard depth-first left-to-right computation rule is applied. The search rule however is not taken into account. This implies that the order of child nodes of an AND-node is fixed, whereas for an OR-node their order is irrelevant. Independence of the search rule allows certain optimisations, such as analysing the non-recursive clauses in a predicate definition before the recursive ones. The construction of the AND-OR-graph is performed by the application independent abstract interpretation procedure. This procedure is basically a fixpoint algorithm which computes the abstract semantics of the program. It is defined in terms of three abstract operations that are defined in section 3.2.3. An efficient version of the abstract interpretation procedure, incorporating several optimisations, has been developed [94].
In the rest of this chapter, we first describe the requirements on the abstract domain. Then we give a general specification of the abstract operations and formulate their safety conditions. We mainly follow [43, 41], but put more emphasis on the *primitive* abstract operations. Finally, we outline the abstract interpretation procedure.

3.2.2 Concrete and abstract domain

In the case of CLP, the concrete domain, denoted Con^c, contains sets of constraints. We use CS (with or without subscript) to denote a set of concrete constraints (an element of Con^c) and C (with or without subscript) to denote a single concrete constraint (an element of $Cons$). The order relation $\leq^c$ on Con^c is defined to be set inclusion.
The elements of the abstract domain Con^a are called *abstract constraints*. An abstract constraint describes a set of concrete constraints, via the concretisation function γ, and is denoted by AC (with or without subscript) in the sequel. Although the framework does not refer explicitly to the abstraction function α, we assume such a function is given. Note that, in some cases, it may be more straightforward to define α instead of γ. If $(Con^c, \leq^c) \underset{\alpha}{\overset{\gamma}{\rightleftarrows}} (Con^a, \leq^a)$ is a Galois connection, then all framework conditions can be reformulated in terms of α. The abstract domain Con^a can be split up into a number of subsets Con^a_D, where the elements of Con^a_D are abstract constraints over the same *domain* D of variables (variables of the clause or query they adorn). Similarly, Con^c can be split up into different Con^c_D and γ and α provide a linkage between elements of Con^a_D and Con^c_D. Each Con^a_D is required to have a minimal algebraic structure. There must be :

1. a preorder $\leq^a$ satisfying $\forall AC_1, AC_2 \in Con^a_D : AC_1 \leq^a AC_2 \Rightarrow \gamma(AC_1) \leq^c \gamma(AC_2)$. This preorder induces an equivalence relation $\equiv^a$ on Con^a_D as follows $\forall AC_1, AC_2 \in Con^a_D : AC_1 \equiv^a AC_2 \iff AC_1 \leq^a AC_2 \ \& \ AC_2 \leq^a AC_1$.

2. an upper bound operator $upp : Con^a_D \times Con^a_D \rightarrow Con^a_D$ such that $\forall AC_1, AC_2 \in Con^a_D : AC_1 \leq^a upp(AC_1, AC_2) \ \& \ AC_2 \leq^a upp(AC_1, AC_2)$. This operator can easily be generalised to compute the upper bound of more than two abstract constraints. In the sequel we assume that upp is defined on a set of abstract constraints.

3. a maximal element $AC_{max} \in Con^a_D$ such that $\gamma(AC_{max}) = Cons_D$ and $\forall AC \in Con^a_D : AC \leq^a AC_{max}$.

4. a minimal element $\bot \in Con^a_D$ such that $\gamma(\bot) = \emptyset$ and $\forall AC \in Con^a_D : \bot \leq^a AC$.

5. a widening operator $W : Con^a_D \times Con^a_D \rightarrow Con^a_D$ [27] such that
   - $W(AC_1, AC_2) \geq^a AC_1$ and $W(AC_1, AC_2) \geq^a AC_2$ for all $AC_1, AC_2 \in Con^a_D$ and
   - for all ascending chains $AC_1 \leq^a AC_2 \leq^a \ldots$ with $AC_i \in Con^a_D$, the ascending chain defined by $AC'_1 = AC_1, \ldots, AC'_{i+1} = W(AC'_i, AC_{i+1}), \ldots$ is not strictly ascending.

   Note that if Con^a_D is finite or has no infinite ascending chains for $\leq^a$, W can simply be defined as *upp* (a stronger definition of W may nevertheless be useful to accelerate termination of the analysis).

According to condition 1, it is allowed to have different descriptions for the same set of constraints; however, the equivalence classes induced by the preorder have to be partially ordered. Concerning condition 2, it is desirable to define *upp* as precise as possible. If the abstract domain is equipped with a complete partial order, the optimal *upp* is the least upper bound. Condition 3 ensures that every set of constraints has an abstraction. Condition 4 provides an initial value for starting a fixpoint computation in the abstract domain. It also provides a precise abstraction to adorn unreachable program points. Finally, condition 5 ensures the existence of a widening operator which can enforce a safe approximation of a fixpoint in a finite number of steps.

The above requirements are weaker than the ones that are used in the traditional framework of Cousot&Cousot [23] based on Galois connections. Still, having a Galois connection is a valuable property : it ensures that the abstractions are not only safe, but also that for each element of the concrete domain there exists a unique *best* (i.e. most precise) abstraction in the abstract domain, to which it should be mapped by α. The determination of a best or most precise abstraction is based on a so called *approximation order* $\preceq^a$ [26, 25] on the abstract domain. This order describes the relative precision of abstract descriptions. E.g. consider the abstract domain $Dom^a = \{\bot, 0, +, -, \top\}$ where $\gamma(\bot) = \emptyset$, $\gamma(0) = \{0\}$, $\gamma(+) = \{n \in \mathbb{Z} \mid n \geq 0\}$, $\gamma(-) = \{n \in \mathbb{Z} \mid n \leq 0\}$ and $\gamma(\top) = \mathbb{Z}$; then the approximation order $\preceq^a$ is defined as follows : $\bot$ is the minimal element, $\top$ is the maximal element, $0 \preceq^a +$ and $0 \preceq^a -$. In most cases (as is also assumed in the above framework description), the approximation order on the abstract domain coincides with the so called *computational order* [26, 25] on the abstract domain; the latter is the order between successive iterates in the fixpoint computation (i.e. between successive abstractions at a program point). In general however, the approximation and computational orders are distinct [26, 25] (see Section 5.2.4 for a distinction between these orders on the concrete domain).

3.2.3 Abstract operations and safety conditions

This section specifies the abstract operations and defines the associated safety conditions that guarantee the safety of the overall analysis.
In the original description of the framework of Bruynooghe [6], the abstract operations are *procedure-entry*, *procedure-exit* and *abstract unification*. In the context of CLP, two modifications are introduced. First of all, abstract unification is replaced with *abstract interpretation of a constraint*. Secondly, two *primitive abstract operations* are identified in terms of which the three larger abstract operations may be defined. These primitive operations are *abstract conjunction* and *abstract projection*. These modifications have first been

described by García de la Banda and Hermenegildo in [43] (although with less emphasis on the distinction between primitive and other abstract operations).
Initially, we will assume that input programs are *normalised*, i.e. the arguments of literals and clause heads are distinct variables. This means that all equality constraints are made explicit. As a consequence, we can simplify parameter passing to variable renaming. The real work is done during the explicit constraint solving. As in [59, 91] and in contrast to the original framework [6], the normal form allows to simplify the definitions and proofs of the abstract operations. Eventually, the analysis of non-normalised programs will be discussed in Chapter 9.

3.2.3.1 Primitive abstract operations

1. Abstract conjunction

The abstract conjunction operation mimics the concrete conjunction of two sets of constraints. Given $CS_1, CS_2 \in Con^c$, the concrete conjunction of CS_1 and CS_2 is the set of pair-wise conjunctions of the constraints in CS_1 and CS_2 :

$$CS_1 \wedge CS_2 = \{C_1 \wedge C_2 \mid C_1 \in CS_1, C_2 \in CS_2\}.$$

For $AC_1, AC_2 \in Con^a$, we let $AC_1 \wedge AC_2$ denote the abstract conjunction of AC_1 and AC_2. The domain of $CS_1 \wedge CS_2$ (resp. $AC_1 \wedge AC_2$) is the union of the domains of CS_1 and CS_2 (resp. AC_1 and AC_2).

SAFETY CONDITION :
Let $AC_1, AC_2 \in Con^a$. For all $CS_1, CS_2 \in Con^c$ such that $CS_1 \leq^c \gamma(AC_1)$ and $CS_2 \leq^c \gamma(AC_2)$ must hold that $CS_1 \wedge CS_2 \leq^c \gamma(AC_1 \wedge AC_2)$.

The above can also be written as : for all $C_1, C_2 \in Cons$ such that $C_1 \in \gamma(AC_1)$ and $C_2 \in \gamma(AC_2)$ must hold that $C_1 \wedge C_2 \in \gamma(AC_1 \wedge AC_2)$ (emphasising a single constraint rather than a set of constraints)[3].

If $(Con^c, \leq^c) \underset{\alpha}{\overset{\gamma}{\rightleftarrows}} (Con^a, \leq^a)$ is a Galois connection, this condition can be replaced by an equivalent one in terms of α and $\leq^a$ (instead of γ and $\leq^c$) :
for all $CS_1, CS_2 \in Con^c$ such that $\alpha(CS_1) \leq^a AC_1$ and $\alpha(CS_2) \leq^a AC_2$ must hold that $\alpha(CS_1 \wedge CS_2) \leq^a AC_1 \wedge AC_2$.

2. Abstract projection

The abstract projection operation mimics the concrete projection of a set of constraints onto a set of variables. It restricts attention to those variables. The concrete projection of $CS \in Con^c$ onto $V \subseteq Var$ is denoted $\overline{\exists}_V CS$ and is defined as follows :

$$\overline{\exists}_V CS = \{\overline{\exists}_V C \mid C \in CS\}.$$

Given $AC \in Con^a$ and $V \subseteq Var$, we let $\overline{\exists}_V AC$ denote the abstract projection of AC on V. The domain of $\overline{\exists}_V CS$ and $\overline{\exists}_V AC$ is V.

SAFETY CONDITION :
Let $AC \in Con^a$.
For each $CS \in Con^c$ such that $CS \leq^c \gamma(AC)$ must hold that $\overline{\exists}_V CS \leq^c \gamma(\overline{\exists}_V AC)$.

Again, when dealing with a Galois connection, the following formulation is equivalent :
for each $CS \in Con^c$ such that $\alpha(CS) \leq^a AC$ must hold that $\alpha(\overline{\exists}_V CS) \leq^a \overline{\exists}_V AC$.

[3] The same can be done for all safety conditions in the rest of this chapter.

3.2.3.2 Abstract operations

Each of the abstract operations defined below updates the AND-OR-graph and computes an abstract constraint in some program point. The AND-OR-graph is considered as a global data structure for which the updating is an implicit side-effect of the operations.

In the sequel, the following naming conventions are used :

- AC_c and AC_s denote respectively the abstract call and the abstract success constraint of a literal (the domain of AC_c and AC_s is the set of variables in the clause or query containing the literal).
- AC_{entry} is the projection of the abstract call constraint AC_c of an atom A onto $vars(A)$, AC_{exit} (also with domain $vars(A)$) is the answer constraint of A.
- AC_{in} and AC_{out} are the abstract constraints at the beginning and at the end of a clause Cl, respectively. (Note that AC_{in} is also the abstract call constraint of the first literal in the body of Cl and AC_{out} is the abstract success constraint of the last literal in Cl.) The domain of AC_{in} and AC_{out} is $vars(Cl)$.

The same conventions are applied when naming sets of concrete constraints.

1. `Procedure-entry`$(A, AC_c) = \{AC^1_{in}, \ldots, AC^m_{in}\}$

The procedure-entry operation takes as input an atom A and its abstract call constraint AC_c. For each clause[4] Cl^j defining A, it computes the abstract constraint AC^j_{in} to be attached to the first program point in Cl^j. The computation consists of two steps. First, AC_c is projected onto the variables of the call A, yielding AC_{entry} (more formally, $AC_{entry} = \exists_{vars(A)} AC_c$ using the abstract projection operation described above). Secondly, AC^j_{in} is obtained from AC_{entry} by matching the call A with the head H^j of Cl^j and by adding initial information for the local variables in the clause Cl^j, as the domain of AC^j_{in} is defined to be the set of variables in Cl^j. If the input program is normalised, matching A with H^j (i.e. $A = H^j$) is reduced to a simple variable renaming, which maps the arguments of A to the corresponding arguments of H^j.

As a side-effect, procedure-entry extends the AND-OR-graph at the OR-node for A, by adding a branch for each clause Cl^j ($Cl^j \equiv H^j \leftarrow B^j_1, \ldots, B^j_{k_j}$) as shown in Figure 3.1.

SAFETY CONDITION :
Let $\rho \in Ren$ which maps the arguments of A to the corresponding arguments of H^j. Then

1. for each $CS_c \in Con^c$ such that $CS_c \leq^c \gamma(AC_c)$ must hold that $\exists_{vars(A)} CS_c \leq^c \gamma(AC_{entry})$;
2. for each $CS_c \in Con^c$ such that $\exists_{vars(A)} CS_c \leq^c \gamma(AC_{entry})$ must hold that $(\exists_{vars(A)} CS_c)\rho \leq^c \gamma(AC^j_{in})$.

The first part of the safety condition is nothing else then the safety condition of abstract projection. An equivalent formulation of the condition in terms of α can be given when dealing with a Galois connection :

[4] Cl^j is assumed to be renamed apart from the variables in A and AC_c.

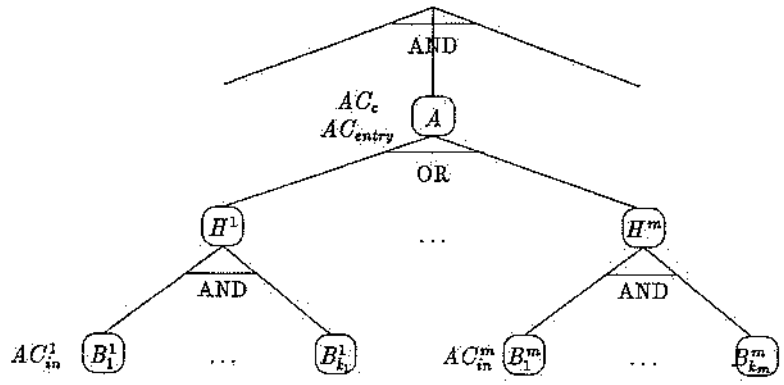


Figure 3.1: Procedure-entry

1. for each $CS_c \in Con^c$ such that $\alpha(CS_c) \leq^a AC_c$ must hold that $\alpha(\overline{\exists}_{vars(A)} CS_c) \leq^a AC_{entry}$;
2. for each $CS_c \in Con^c$ such that $\alpha(\overline{\exists}_{vars(A)} CS_c) \leq^a AC_{entry}$ must hold that $\alpha((\overline{\exists}_{vars(A)} CS_c)\rho) \leq^a AC^j_{in}$.

2. `Procedure-exit`$(A, AC_c, \{H^1, \ldots, H^m\}, \{AC^1_{out}, \ldots, AC^m_{out}\}) = AC_s$

Procedure-exit is illustrated in Figure 3.2. It can be applied when the final abstract constraint AC^j_{out} has been derived for each clause $Cl^j : H^j \leftarrow B^j_1, \ldots, B^j_{k_j}$ defining A. The procedure-exit operation takes as input an atom A, its abstract call constraint AC_c, the heads H^j of the clauses defining A and the set of abstract constraints $\{AC^1_{out}, \ldots, AC^m_{out}\}$. It computes the abstract success constraint AC_s of A. In a first step, AC_{exit} is computed which has as domain the set of variables of A. More precisely, $AC_{exit} = upp(\{AC^1_{exit}, \ldots, AC^m_{exit}\})$ where AC^j_{exit} is obtained by (1) projecting AC^j_{out} onto the variables of H^j and (2) performing backward matching $H^j = A$ (if the program is normalised, this simplifies to variable renaming). The second step is the so called *extension* step which combines AC_{exit} and AC_c to obtain AC_s; it computes the effect of executing the procedure (which resulted in AC_{exit}) onto the variables of AC_c that do not occur as arguments of A (recall that the domain of AC_c is the set of *all* variables in the clause in which A occurs, so it includes the domain of AC_{exit} which is just $vars(A)$).

SAFETY CONDITION :

Let $\rho \in Ren$ which maps the arguments of A to the corresponding arguments of H^j; ρ^{-1} denotes the reverse mapping.

1. For each $CS^j_{out} \in Con^c$ such that $CS^j_{out} \leq^c \gamma(AC^j_{out})$ must hold that $CS_{exit} \leq^c \gamma(AC_{exit})$ with $CS_{exit} = (\overline{\exists}_{vars(H^j)} CS^j_{out})\rho^{-1}$ and $1 \leq j \leq m$.
2. For each pair $CS_c, CS_{exit} \in Con^c$, such that $CS_c \leq^c \gamma(AC_c)$ and $CS_{exit} \leq^c \gamma(AC_{exit})$ and there exists a $CS_{local} \in Con^c$ over $vars(A)$ with[5] $CS_{exit} = (\overline{\exists}_{vars(A)} CS_c) \wedge CS_{local}$, must hold that $CS_s \wedge CS_{exit} \leq^c \gamma(AC_s)$.

[5] CS_{local} is the set of all possible constraints – projected onto $vars(H^j)$ and renamed to $vars(A)$ – that

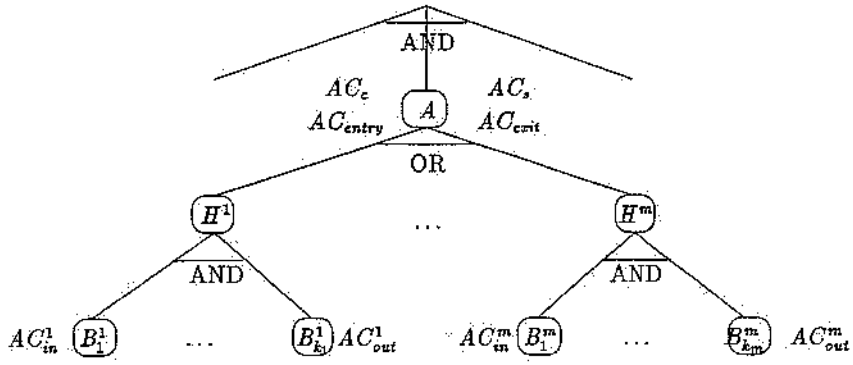


Figure 3.2: Procedure-exit

An equivalent formulation in terms of α can be given when dealing with a Galois connection.

Note that AC_s can be defined in terms of the abstract conjunction operation : $AC_s = AC_c \wedge AC_{exit}$. The computation of AC_{exit} is based on the abstract projection and *upp*. As a consequence, the safety of procedure-exit then depends on the safety of abstract conjunction and abstract projection and on the properties of *upp*.
Sometimes, more precision can be obtained by defining AC_s *not* directly as $AC_c \wedge AC_{exit}$. This is the case for the analyses that will be presented in Chapters 5, 6, 7 and 8. However, abstract conjunction can then be applied on components of the abstract constraints; so the safety of abstract conjunction still largely determines the safety of procedure-exit.

3. `Abstract-interpretation-constraint`$(C, AC_c) = AC_s$

The abstract interpretation of a constraint takes as input a constraint C and its abstract call constraint AC_c and produces its abstract success constraint AC_s. The operation mimics the concrete conjunction of the constraint C to the current store.

SAFETY CONDITION :
For each $CS \in Con^c$ such that $CS \leq^c \gamma(AC_c)$ must hold that $CS \wedge \{C\} \leq^c \gamma(AC_s)$.

An equivalent formulation in terms of α can be given when dealing with a Galois connection.

Again, AC_s can be defined in terms of the abstract conjunction operation : $AC_s = AC_c \wedge \alpha(C)$. Safety of the abstract conjunction therefore implies the safety of the abstract interpretation of a constraint. Note that, although this definition satisfies the safety condition, it does not necessarily yield the best (most precise) result, since $\gamma(\alpha(C))$ may be larger than $\{C\}$.

---

may be gathered during execution of Cl^j (each constraint is gathered along some OR-branch of the concrete AND-OR tree). Alternatively, the condition $CS_{exit} = (\exists_{vars(A)} CS_c) \wedge CS_{local}$ can also be formulated as follows (without using CS_{local}): $CS_{exit} \Rightarrow \exists_{vars(A)} CS_c$.

3.2.4 Abstract interpretation procedure

The abstract interpretation procedure is the application independent kernel of the framework. It is used to construct the AND-OR-graph. Basically, it is a fixpoint algorithm computing an abstraction of the LD semantics (SLD semantics, with selection of the leftmost atom as computation rule). It is convenient to assume that queries consist of a single literal; this does not incur any loss of generality. The abstract AND-OR-graph for a set of queries, described by the literal Q and the abstract call constraint AC, is initialised with the root node representing Q and adorned to the left with AC. The recursively defined abstract interpretation procedure is then called for the initial AND-OR-graph : abstract-interpretation(Q, AC).

Algorithm 3.2.1 (abstract-interpretation(L, AC_c))
Input : *A program, a literal L and its abstract call constraint AC_c.*
Output : *The abstract success constraint AC_s.*
If $AC_c = \bot$ then $AC_s = \bot$, otherwise three cases can be distinguished :

Case 1 : L is a constraint. Then call abstract-interpretation-constraint(L, AC_c).

Case 2 : L is an atom and L has no ancestor node in the partially constructed AND-OR-graph with a call to the same predicate. Then, if there exists at least one clause defining L in the program,

- call procedure-entry(L, AC_c);
- call abstract-interpretation for each of the literals in the body of each clause $Cl^j : H^j \leftarrow B^j$ defining L in a left-to-right order;
- call procedure-exit($L, AC_c, \{H^1, \ldots, H^m\}, \{AC^1_{out}, \ldots, AC^m_{out}\}$).

Otherwise (if there is no clause defining L in the program), $AC_s = \bot$.

Case 3 : L is an atom and L has an ancestor node in the partially constructed AND-OR-graph with a call L' to the same predicate; let AC'_c be the abstract call constraint of L'. Let $AC_{entry} = \exists_{vars(L)} AC_c$ and $AC'_{entry} = \exists_{vars(L')} AC'_c$.

1. If $AC_{entry} \equiv^a AC'_{entry}$ up to renaming of variables (i.e. both atoms have the same entry pattern), then AC_{exit} is a renaming of AC'_{exit} which is computed by the first step of procedure-exit($L', AC'_c, \{H^{1'}, \ldots, H^{m'}\}, \{AC^{1'}_{out}, \ldots, AC^{m'}_{out}\}$), using $\bot$ as abstract success constraint for the branches j for which $AC^{j'}_{out}$ is not yet computed. Next, AC_s is computed from AC_c and AC_{exit} via the extension step (second step in procedure-exit).

   At some point, procedure-exit will be called for L' and a new value will be obtained for AC'_{exit}. Let AC_1 be the old and AC_2 be the new value. If $AC_2 \leq^a AC_1$, then the computation proceeds as usual with the old value AC_1 and node L refers back to ancestor node L'; otherwise the computation starts over at the call L using $W(AC_1, AC_2)$ as new value for AC_{exit} and AC'_{exit}.

2. If $AC_{entry} \not\equiv^a AC'_{entry}$ up to renaming of variables, then the computation can proceed as in Case 2 if the abstract domain is finite. Otherwise, the depth of recursion has to be controlled e.g. by setting a threshold. If the threshold is reached, the following is done : if $AC_{entry} \leq^a AC'_{entry}$, then the computation proceeds as in Case 3.1; otherwise, the subgraph of L' is recomputed using $W(AC'_{entry}, AC_{entry})$ as new value for AC'_{entry} (used in the second step of procedure-entry). Instead of using a fixed threshold, a more sophisticated specialisation criterion could be used to decide whether to continue the computation for L and AC_{entry} as in Case 2 (i.e. to analyse another predicate version) or not.

Chapter 4

Related work

In this chapter we discuss related work on semantics and abstract interpretation of CLP, on CLP optimisations and analyses, and on LP mode analyses. We also situate our work in this context and summarise its most important characteristics.

4.1 Semantics and abstract interpretation of CLP

4.1.1 Frameworks for abstract interpretation of CLP

Abstract interpretation of logic programs has been an active area of research (e.g. [6, 25, 33, 31, 82, 70, 89, 97]). A large number of frameworks and applications have been developed and were shown to be useful in optimising the compilation of logic programs [72, 33, 112, 110, 107]. As abstract interpretation is also expected to be useful in the context of CLP [63, 55, 85], a few general frameworks have already been defined to that aim. They are based on the semantics of CLP (an extensive description of the different semantics for CLP is given in [55]).

Marriott and Søndergaard [83] present a general and elegant framework built on a meta-language that can express the semantics of a wide variety of programming languages, including CLP languages. The meta-language is based on the formalism of denotational semantics. It can easily express both standard and non-standard semantics. However, from a practical point of view, this framework does not much simplify the development of an abstract interpretation system. It is a high-level description, whereas an actual system is based on a more elaborated abstract interpretation procedure that incorporates clearly identified domain-dependent operations and associated safety requirements. Such a more extensive specification is given by Bruynooghe's framework [6]), whose adaptation for CLP is described by García de la Banda and Hermenegildo [43, 41]. It forms the basis of our analyses in subsequent chapters.

In [62], Jørgensen presents a somewhat modified version of the denotational framework for abstract interpretation of Marriott and Søndergaard, and designs an algorithm for analysis of CLP programs within that framework. Correctness and complexity results for the algorithm are given.

Codognet and Filé [17] also present a general framework that encompasses both the execution of (constraint) logic programs and their static analyses. Their framework is based on the notion of a *computation system*, which is a 5-tuple $S = \langle \Sigma, I, D, \oplus, \pi \rangle$ where Σ is

an alphabet of function and predicate symbols, I is an interpretation of the symbols in Σ, D is a computation domain, $\oplus$ is a composition function (composition in the sense of conjunction) and π is a projection function. They also define the notion of simulation or abstraction between two computation systems S and S'. Abstract S'-programs perform the abstract interpretation of the concrete S-programs. The framework naturally accommodates multiple layers of abstraction, where one computation system S is abstracted by another one S' which is again abstracted by S'' and so on. Although being more concrete than the approach of Marriott and Søndergaard, this proposal still has some shortcomings on the level of practicality. For example, it lacks a definition for the order and equivalence of sets of constraints (which involves the definition of some kind of canonical form for constraints), and a join or upper bound operation in order to gather the results of different computation paths. On the other hand, the paper introduces the quite interesting idea of implementing the abstract operations within a CLP language themselves. Thus static analysis of concrete programs in some CLP language is performed by running corresponding abstract programs in the same or another CLP system, thereby exploiting the constraint solvers in that system. For example, a definiteness analysis of CLP(H,N) programs can be obtained by executing CLP(Boole) programs, using *Prop* [81, 84, 21] (a subset of the propositional formulae) as abstract domain. Termination of the computation is ensured by means of a tabulation mechanism.

In [45, 46], Giacobazzi, Debray and Levi describe a general algebraic framework for CLP that is parameterised with respect to the underlying constraint system. Both the operational (top-down) and fixpoint (bottom-up) semantics of CLP can be formulated within this framework. Abstract interpretation of CLP programs is viewed as just another instance of the general framework, in the same way as the concrete semantics that it approximates. Abstract computation is specified as a standard CLP computation over an appropriate non-standard constraint system. Note that, in some cases, the non-standard constraint system is just another constraint system part of some CLP language, e.g. definiteness analysis of CLP(H,N) can be performed by executing CLP(Boole) programs. In contrast to the approach of Codognet and Filé, no tabulation is considered which makes the semantics construction more general; finiteness is just a specific property of the constraint system. Also, the algebraic structure of a computation system considered by Codognet and Filé is quite different: only composition in the sense of conjunction is considered there, no join operation is provided. The algebraic approach to constraint interpretation makes it easy to identify a suitable set of operators, which can be instantiated in different ways to obtain the definition of different non-standard semantics. Within an algebraic setting, the notion of abstraction can be characterised by weakening constraints, and relationships between different abstract interpretations can be characterised in terms of homomorphisms between the corresponding algebras. Similarly to the approaches mentioned above, this work is also in fairly general terms and does not offer much support to the application developer.

Finally, Bruynooghe and Janssens [11] present a framework for abstract interpretation of CLP that extends Bruynooghe's framework for LP [6]. It is based on the idea of adding complexity to the framework which may possibly decrease the complexity in the abstract domain. This is done by incorporating the *repeat-previous-call* strategy (introduced in [6], and mentioned by Le Charlier and Van Hentenryck [71] under the name *reexecution* strategy). Some goals or constraints that have already been analysed can be reconsidered at a later program point; this is useful if their execution can provide further information in com-

bination with the new information derived since the goal or constraint was first analysed. In this way, reexecution allows to improve the precision of the analysis without having to keep track of complex information, such as variable dependency or sharing information, within the abstract domain. Alternatively, one could characterise the framework by stating that the depth first computation rule is replaced by a subtle coroutining : the computation of a predicate can suspend and continue with the execution of another predicate or constraint. A similar situation appears when considering parallel execution of logic programs, or when analysing concurrent logic programs.

As already pointed out in [43, 41], a common characteristic of the above frameworks is that they more or less depart from the approaches that have been successful in analysing traditional LP languages. The framework introduced in [43, 41] (presented in Section 3.2) is a quite straightforward extension of one of the frameworks for LP, more precisely the framework of Bruynooghe [6]. This framework led to several practical variants [94, 70] for which there exist efficient fixpoint algorithms. These algorithms can be reused in the context of CLP. Hence, developing a particular analysis within that framework is reduced to specifying and implementing the domain-dependent operations and safety requirements.

4.1.2 Combining abstract domains

Part of the work described in this thesis consists of combining two analyses for CLP : a freeness and a definiteness analysis. The idea is to obtain a full mode analysis system and to get a more compact and more efficient freeness abstraction while maintaining its precision. An approach for combining abstract domains is described by Cousot and Cousot in [24] and is applied in the context of LP by Codish et al. [16]. This approach does not change the abstractions and keeps the original components of the basic operations. During analysis, interactions occur to refine the abstractions. This results in a precise combined analysis, in particular when the analyses being composed contain a sufficient degree of "overlapping" information. Our approach is different in the sense that the freeness abstraction is redefined in terms of a definite and free component, yielding a more efficient but equivalent freeness abstraction. The definite component is an integral part of the improved freeness abstraction. For this abstraction, new definitions have to be given in terms of the two components. However, it is possible to retain important basic operations such as abstract conjunction.

Recently, Cortesi, Le Charlier and Van Hentenryck [22] have presented a new approach to combine abstract domains (applicable to but independent from LP). They define the notion of *open product* of domains that allows each combined domain to benefit from information in the other domains through the notions of queries (providing information to the environment) and open operations (receiving information from the environment). The combination of the definiteness and freeness analysis for CLP described in this thesis could be defined as an application of this methodology : a query on the definiteness domain can provide information on the definiteness of variables, which can then be used by the open operations on the freeness domain. However, in the approach of [22], operations can only benefit from information *before* the product operation or can be refined *after* the product using the resulting information in each domain. In our combined analysis, another kind of interaction occurs, which can be viewed as a generalisation of the open product definition given in [22] : the *resulting* information of a definiteness operation is used *during* the

corresponding freeness operation (instead of first performing the freeness operation using only the definiteness information that is available before the product operation, and then refining it afterwards using the resulting definiteness information, which is less efficient).
The paper [22] also introduces a generic pattern domain that automatically upgrades an abstract domain with structural information, resulting in a more precise abstraction. The integration of structural information is not studied in this thesis. The extension of our analysis to non-normalised programs already improves the precision of the analysis by taking into account local structural information; of course, keeping track of global information is still more powerful and yields more precise results. It is not yet clear however how the addition of structural information would influence the complexity and efficiency of the analysis. The size of our abstractions could be reduced in some cases, but there is the additional structure component. One also has to deal with the interaction of this component with the rest of the abstraction. Integrating structural information is an interesting topic for further research.

4.2 CLP optimisations

Before going into the specific analyses developed within the abstract interpretation frameworks, we first recite the possible optimisations of CLP programs that gave rise to these applications. A summary of most of these optimisations can also be found in [85]; this paper further contains examples and experimental results obtained by applying one or a combination of the optimisations. Exactly which optimisations are possible depends on how run-time structures are handled by an implementation. However, many of the optimisations can be understood as source-level transformations and require only limited knowledge about an implementation.

Constraint specialisation

A low-level optimisation is constraint specialisation (also referred to as bypassing the constraint solver [63, 85]). The by now well-known specialisation of unification [107, 110, 77] is just a subcase; for more details about it we refer to [107, 110, 77]. Specialisation is even more useful in the case of numerical constraints. The idea is to detect when constraints or parts of constraints are trivial in the sense that simple numerical evaluation, testing and/or assignment suffice, thus avoiding the invocation of the numerical solvers. In particular, by specialising inequalities into simple tests the expensive simplex algorithm can be avoided, yielding significant speed-ups as was shown e.g. in [63, 85]. Also, the size of the constraint store decreases. Moreover, by performing the specialisation at compile-time, the assignment or testing can be hardwired in the generated code.
In [10], we investigated the feasibility of extracting specialised predicate versions based on a transformation technique called *compiling control*. This technique was originally devised for transforming logic programs requiring a special computation rule into Prolog programs [29]. The idea of the method is, given a rule for constraint activation that is based on mode information, to transform a CLP program into a specialised one in which constraints are transformed into simple tests or assignments.
In order to perform specialisation, mode information is required. Mode information should include at least freeness and definiteness information (sometimes also a limited form of

type information is included [63]). Freeness is defined as follows : a variable is *free* if it is unconstrained except for possible bindings with other free variables. This is the most restrictive definition of freeness; in Chapter 5.1 we will introduce another less restrictive notion that is useful for other optimisations such as constraint reordering. A variable is said to be *definite* if it is constrained to a unique value.

Designing specialised constraint solver instructions

Even if the constraint solver still has to be called, some special cases of constraint solving stand out [55] :

- The constraint is to be added to the store, but no satisfiability check is needed.
- The constraint need not be added, but its satisfiability in conjunction with the store needs to be checked. Of concern here are constraints that can be shown to become redundant as a result of future additions to the store. This notion of *future-redundancy* was first described in [63].
- The constraint needs to be added and satisfiability needs to be checked, but the constraint is never used later. Variables in the constraint are never referred to in the remaining computation. This optimisation was introduced under the name *dead variable removal* in [75].

As the second and third case are quite important optimisations, they are discussed in more detail below. The special cases justify making specialised versions of the constraint solver instructions. In the case of CLP($\mathcal{R}$), Jaffar et al. [56] and also Marriott et al. [85] consider code generation for an abstract machine, CLAM, whose instruction set includes instructions designed specifically with an optimising compiler in mind.

Removal of (future) redundant constraints

One of the most prominent optimisations for CLP, as shown by the experimental results in [63, 85], is the removal of (future) redundant constraints. A constraint is called *redundant* if it is entailed by other constraints in the current store. It is said to be *future redundant* if it will become redundant in a future constraint store. Jørgensen et al. [63] restrict the future constraint store to the one obtained by joining the constraints in the next cycle of the constraint engine with the current store; Marriott and Stuckey [86] generalise this to any future constraint store, obtained through any number of derivation steps starting from the current store.

If a constraint is known to be redundant by the time it is encountered, it is certainly consistent with the store at that point and it does not have to be added. So, such a constraint can simply be removed from the program. A future redundant constraint still has to be checked for satisfiability in conjunction with the current store. Depending on the exact definition of future redundancy, two cases can be considered :

- When considering the more restrictive definition of Jørgensen et al. [63], a future redundant constraint in this sense still has to be checked for satisfiability in conjunction with the current store, but it does not have to be added to the store.
- According to the more general definition of future redundancy given by Marriott and Stuckey [86], a program can be optimised by adding removal instructions at those

points where initially non-redundant constraints will have become redundant. Such a removal command for a constraint can be reordered (more precisely, can be moved towards the beginning of the clause in which it appears), if the constraint cannot cause failure over the period between its set-up and removal. This may lead to a special case where the removal instruction immediately follows the constraint; this is exactly the case described by Jorgensen et al.

In general, the removal of (future) redundant constraints is based on detecting definite interaction between constraints. For the more restrictive subcase, even no global dataflow analysis is needed (although predicate call information may increase the detection of future-redundancy and hence the applicability of the optimisation). A simple method consists of unfolding the predicate definition (typically once is enough), and then to determine whether the constraints in the unfolded definition in conjunction with the local constraints (except for the constraint c that is checked for future redundancy) imply c.

Removal of dead or redundant variables

Another source of redundancy in the constraint solver are variables which will never be referred to in the remaining computation, although they are still continually manipulated. These variables are called *dead* variables [75]. Their continuous manipulation causes a considerable slow-down. It also gives rise to large constraint stores. An example of this case will be given in Chapter 11.
Execution can be improved by adding instructions which project out the dead variables. Clearly this requires determining which variables may still be referred to in the remaining computation. However, some care has to be taken when applying dead variable removal, as it interferes with the form in which constraints are stored by the system. For the moment, consider only linear equations. Removing a non-parametric[1] variable can be performed quite easily. It simply consists of removing the equation that contains the variable as its left-hand side. However, the removal of parametric variables is effected by pivoting the representation of the stored constraints in order to make the variable non-parametric. The cost is similar to adding a new equation. Moreover, deleting or changing an equation also interferes with backtracking. So the overhead of removing a parametric variable may possibly outweigh the gain of the removal. Therefore the strategy suggested by Macdonald et al. [75] is to eliminate only non-parametric variables in equations, and not to consider elimination of parametric variables in equations nor of variables in equalities or non-linear constraints.

Constraint/goal reordering

Reordering of constraints is studied by a number of authors. Marriott and Stuckey [86] discuss reordering within a general setting of techniques applicable in the compilation of CLP languages (the collection of techniques presented there are *refinement*, *constraint removal* and *reordering*). Reordering consists of moving constraint addition later and (future) redundant constraint removal earlier in a program clause when the constraint involved cannot affect the execution in the intervening computation (i.e. cannot cause unsatisfiability).

[1] A definition of parameters and non-parameters in the solved form of constraints has been given in Section 2.2.

The underlying idea is that a constraint should generally be introduced as late as possible in a computation, and should be removed as soon as it has become redundant; this allows to reduce the number of constraints in the store and hence also the execution time. García de la Banda, Hermenegildo and Marriott [44] also consider the later addition of constraints. They refer to it as (repeatedly) reordering a (sub)goal of the form c, g to g, c where c is a primitive constraint and g is a user-defined goal. They also lift reordering to goal reordering (reordering of g_1, g_2 to g_2, g_1).
Reordering requires determining possible interaction between constraints/goals. If constraints/goals do not interact, they are called independent. Several types of independence are defined in [44].

AND-parallelisation

Information on constraint/goal independence can also be used to exploit independent AND-parallelism. García de la Banda et al. [44] extend the results obtained for LP to the more complex case of CLP. The aim is to run in parallel as many goals as possible while maintaining the correctness (same answers) and efficiency with respect to the sequential execution. This can be ensured if particular requirements of goal independence and constraint solver independence are satisfied. The latter basically means that changing the order in which constraints are added should not affect the execution time, or at least should have only a limited influence.

Intelligent backtracking

A third optimisation that can be based on goal independence information is intelligent backtracking. This can improve the efficiency by avoiding reexecution of goals (and therefore of the related constraint solving operations) that have nothing to do with the cause of the failure being handled. Determining intelligent backtracking at run-time may not always yield the expected improvement, as it requires maintaining some kind of history of operations and involves performing run-time tests. On the other hand, inferring the independence information at compile-time [44] such that the compiler can generate specialised code, should be more successful.

Code motion

A standard optimisation technique in conventional logic programming languages [90] is to move loop invariant code outside the loop. In this way the code is executed only once. This technique is referred to as *code motion*. It was studied in the context of LP by a.o. Debray [30] and by Giannotti and Hermenegildo [47]. It can also be applied to CLP programs as indicated in [63].

Refinement

Program refinement is introduced by Marriott and Stuckey [86]. Refinement consists of adding new constraints at the start of program clauses. These constraints will eventually become redundant, so the declarative meaning of the program is unchanged. The detection of redundant constraints is based on computing a weakening of the constraint store at each point. The advantage of refinement is that the new constraints make information available earlier in the computation, and so can improve the operational program behaviour (failure

may be detected earlier, the number of choicepoints may be reduced). Moreover, refined programs may work for a wider class of calls than the original programs, in the sense that the latter may not terminate for particular call patterns. As refinement may change the behaviour of a program, albeit for the better, it is different in nature than the other presented optimisations. It might therefore be performed in an optional pre-compilation stage, in which the programmer can intervene.

Avoiding the delay of passive constraints

Passive constraints are delayed until certain wake-up conditions (involving the variables in the constraints) are satisfied. The idea is that these constraints are too complex to be handled immediately. Managing the delay and wake-up of passive constraints requires special provisions (cf. [55, 56]). This overhead can be avoided by detecting at compile-time that the constraints can be activated immediately when they are encountered. A class of passive constraints are the non-linear numerical constraints. In most of the CLP systems, these constraints are delayed until they become linear through the definiteness of certain variables. If it is known at compile-time that a non-linear constraint will always be linear when encountered during execution, the overhead of dealing with non-linear expressions at run-time can be avoided [56]. The detection of linearity is based on definiteness information.

Mutual exclusion or determinacy

Another optimisation consists of finding out whether the clauses defining a predicate are *mutually exclusive* for a particular entry pattern that indicates the definiteness of some of the arguments. If an argument is free, indexing on that argument must be avoided as all clauses are valid alternatives according to this argument. A CLP system can use information on mutual exclusion to reduce the number of choicepoints for the call, giving both space and time improvements. This optimisation is described in more detail in [63]. It is also known as one of the most important optimisations for LP [107, 110, 77].

4.3 CLP analyses

Recently several analyses for CLP programs have been proposed. Unfortunately, only few of these have been (completely) implemented or even been worked out in detail. The lack of (full) implementations leaves open the practicality of the analyses.

Definiteness analysis for CLP is discussed in a number of papers. This analysis determines which variables are definitely constrained to a unique value.
Marriott and Søndergaard [83] sketch a definiteness analysis as instance of their denotational framework for abstract interpretation of CLP programs. This analysis is general in the sense that it is not applied to a particular constraint domain. As a consequence, it is not very accurate. It does not keep track of definite dependencies[2] between variables, established by the constraints in the program. Hence, it cannot perform precise definiteness propagation.
This analysis can be improved by applying reexecution, as in the extended framework of Bruynooghe and Janssens [11].

[2]Definite dependencies describe for each variable which sets of variables certainly determine its definiteness.

Alternatively, one can improve the abstract domain. In [4], Baker and Søndergaard use the same denotational framework mentioned above to develop a very precise definiteness analysis for CLP($\mathcal{R}$) programs. Definite dependencies between variables are represented by means of positive Boolean functions (propositional formulae). This abstract domain, by now well-known under the name *Prop*, was first introduced by Marriott and Søndergaard [81, 84] and was further studied by Cortesi et al. [21]. The full expressivity of *Prop* is used, including disjunction. This results in a very precise analysis, at least in the absence of non-linear constraints, but worst-case complexity is exponential in the (maximum) number of variables in a program clause. The analysis is currently being implemented in order to investigate its practicality. The authors suggest that also subclasses of *Prop* may be useful for practical program analysis, thereby sacrificing some precision for efficiency.

Codognet and Filé [17] develop a definiteness analysis of CLP programs as an application of their framework for abstract interpretation. By means of example, the analysis is elaborated for CLP($\mathcal{R}$) programs although it is applicable to other constraint domains as well. The abstract domain is again the *Prop* domain. The authors suggest that the practical realisation of the definiteness analysis of CLP($\mathcal{R}$) programs can be performed by executing CLP(Boole) programs with tabulation. However, they do not report on an actual implementation.

In [43], García de la Banda and Hermenegildo present a definiteness abstraction that is based on a high-level description of uniquely constraining patterns, which is easy to obtain for each particular type of constraint domain. These patterns represent the definite dependencies between variables that are required to perform precise definiteness propagation. The domain can be seen as an encoding and implementation of the *Prop* domain, but without disjunction. In this sense the analysis is strictly less precise than the one of Baker and Søndergaard. The collapsing of disjunctive information is done in order to reduce the size of the abstract constraints. The analysis is developed within the framework of Bruynooghe [6] extended towards CLP [43, 41]. It has been implemented within the abstract interpretation system PLAI [94, 96]. We have combined this definiteness analysis with the freeness analyses described in this thesis (cf. Chapters 7 and 8), leading to a more efficient representation of the freeness abstraction and resulting in a full mode analysis system. The implementation of the combination itself was joint work in the context of the ESPRIT project PRINCE. Conceptually, the definiteness analysis shows large similarities to the one of Codognet and Filé and the one of Baker and Søndergaard, as all three are based on the *Prop* domain. It differs in the following aspects : on the level of specification, another framework is used and the analysis is defined in a general setting without restriction to particular constraint domains; on the level of implementation, the analysers are implemented in Prolog, not in a CLP language, hence no constraint solving facilities are available.

Another closely related definiteness analysis has been proposed by Hanus [48]. The main difference with the previous approaches is that this one includes additional information about non-linear constraints. The aim is to detect whether all non-linear constraints in a program eventually become linear at run-time (i.e. no delayed non-linear constraints are left). For that purpose definiteness information is needed (although the definiteness analysis is not the main aim). Definite dependencies between variables are expressed in a *Prop*-like fashion. Similar constructs are used to represent information about non-linear constraints. Abstract constraint solving, which basically corresponds to abstract

conjunction, is performed through a set of normalisation rules. This formulation is less complex than the more direct one given by García de la Banda and Hermenegildo. However, the rules presented in [48] lead to less precise information than is derived by the latter analysis. The precision could be improved by adding further normalisation rules.

A freeness analysis for CLP is described by Marriott and Søndergaard [83]. As for their definiteness analysis, a general definition is given without referring to a specific constraint domain. Possible dependencies between variables, which are needed to perform accurate non-freeness propagation, are not taken into account; this implies that the analysis is not very precise. The notion of freeness is quite restrictive compared to the one we will introduce in Section 5.1 and which was also used in [37, 38, 36]: in [83] a variable is considered to be free as long as it is unconstrained except for aliasing (i.e. binding) with other free variables. Freeness of this kind is required when specialising constraints to assignments, but is too restrictive for optimisations such as constraint/goal reordering. For the latter purpose, our notion of freeness should be used, which states that a variable is free as long as it can still take all possible values that are allowed by its type. Hence, to support a large spectrum of optimisations, both notions of freeness should be available. We will come back to this in Section 9.5.
The papers [37, 38] contain our work on the kernel abstraction for freeness analysis. The notion of freeness applied there relates to determining definite satisfiability of a constraint store, or – stated in an alternative way – to determining possible interactions between constraints. Only the primitive abstract operations (abstract conjunction and abstract projection) are presented. The additional complexity that arises when defining the higher-level abstract operations (in particular procedure-exit) is only briefly addressed. The abstraction is elaborated in Chapter 5 of this thesis.
In [36], we present two optimisations that allow to improve the precision of the freeness abstraction. The first one consists of exploiting definiteness information which is obtained through a combination with the definiteness analysis developed by García de la Banda and Hermenegildo [43]. The second approach consists of retaining only minimal information that safely approximates the original abstraction. As the optimisations are orthogonal they can easily be combined to obtain a practical full mode analysis system. The optimisations are worked out in detail in Chapters 6, 7 and 8 of this thesis.

In [63], Jørgensen, Marriott and Michaylov mention a simple mode analysis for CLP($\mathcal{R}$) that is to be used for constraint specialisation. The set of modes that is considered is $\{Var, Gnd, Num, Arith, Any\}$. The mode *Var* describes variables that are unconstrained except for possible aliasing (binding) with other variables of the mode *Var* (this is the restrictive notion of freeness mentioned above), *Gnd* describes the set of terms that are ground or definite, *Num* describes the subset of the definite terms that are constrained to a unique number, *Arith* indicates that a variable is constrained by one or more numerical constraints (note that *Num* is a subcase of both *Arith* and *Gnd*), and mode *Any* describes any term. So the mode set also incorporates a limited amount of type information. The paper only gives the above mode definitions, the mode analysis itself is not elaborated.

Another group of analyses is more or less oriented towards the numerical constraint domain. Marriott and Stuckey [86] present an abstract domain *LSign* to describe sets of linear equations and inequalities. In [87], this domain is further elaborated and extended towards the treatment of non-linear constraints and unification constraints (i.e. towards the analysis of

full CLP($\mathcal{R}$)), although the extensions are not completely elaborated. The abstraction of a set of linear constraints is obtained by replacing the coefficients in the constraints by their signs. A strictly positive coefficient is replaced by $\oplus$, a strictly negative one by $\ominus$ and zero is kept as such; a special symbol $\top$ represents a coefficient for which there is no precise information and hence may be positive, negative or zero. In order to improve precision, the abstraction of equations is extended with multiplicity information (i.e. information on the number of equations that are represented by an abstract equation). The *LSign* domain gives quite precise information about possible interaction between numerical constraints. Alternatively, it can be described as a "definite degrees of freedom" analysis which determines when sets of constraints are definitely satisfiable. This information allows to detect constraint independence, which forms the basis for optimisations such as constraint/goal reordering, AND-parallelisation and intelligent backtracking. This work is the most closely related work to our freeness analysis. Its major advantage is the enhanced precision, especially for inequalities but also for equations (it keeps track of the constraint symbol and the sign of the coefficients, which are discarded in our analysis). The main deficiencies are that (1) no implementation is reported, such that it cannot be judged whether the efficiency is reasonable, and (2) some aspects that are relevant in order to obtain a complete analyser are not (sufficiently) elaborated (such as procedure-exit, a suitable widening, the order relation and the interaction between the unification and the numerical part). A more elaborate discussion of the analysis and a comparison with our approach will be given in Section 9.2.

A second abstract domain for describing linear equations and inequalities, which is also presented in [86], is the convex hull description *CHull*. It is based on descriptions used by Cousot and Halbwachs [28] for bounds analysis in conventional programming languages. Elements of the abstract domain are sets of linear constraints of the form $s \leq s'$ or $s = s'$. The abstraction of a concrete set of constraints *CS* is the convex hull of the polytopes defined by the constraints in *CS* (strict inequalities in *CS* are hereby relaxed to non-strict inequalities). The convex hull abstraction can be used to determine redundant constraints, i.e. those constraints that are entailed by each concrete constraint in *CS*.

Janssens, Bruynooghe and Englebert [60] present several abstractions for the numerical leaves of terms in CLP(H,N), with increasing complexity and precision. The abstractions are based on intervals which are computed by narrowing rules [73]. An important aspect is the fact that intervals are also used in the abstraction of the numerical components of non-numerical terms, thus integrating both the H and N domain. The kernel abstraction is based on intervals and narrowing rules that have only a local impact. A prototype implementation has been developed and the obtained results are sufficiently precise to recognise (future) redundant constraints. Interval information can also be used to improve the abstraction of non-linear constraints in other analyses (e.g. cf. Section 9.3). The basic abstraction can be extended by incorporating the narrowing rules more globally and, in addition, by keeping track of ordinal relationships between variables (i.e. certain types of inequalities). The extensions maintain numerical constraints in concrete form. The nature of the extensions is the same as for *CHull* : they compute an underestimation or weakening of the concrete set of constraints at each point.

A very limited interval analysis of CLP($\mathcal{R}$) programs is described by Jørgensen [62]. The abstract domain is restricted to the set of abstract descriptions $\{Any, [0, \infty], [1, \infty]\}$. Moreover, intervals are only used to abstract numerical terms (numerical leaves of non-numerical

terms are not considered).
Bagnara, Giacobazzi and Levi [3, 2] integrate constraint propagation techniques into an analysis of CLP programs over numeric domains. The advantage of this approach should be that it allows to close the gap between the formalisation or specification of the analysis and the possibility of an efficient implementation (although no implementation results are reported yet). Constraints are abstracted by means of labelled digraphs. Nodes are labelled with a numerical expression, occurring in a numerical constraint of the program, or with an interval (if the analysis can no longer maintain a precise numerical expression); arcs are labelled with constraint symbols ($=, >, \ldots$). Manipulating the digraphs is done using constraint inference techniques. The analysis allows to detect (future) redundant constraints.

Finally, Macdonald, Stuckey and Yap [75] describe an analysis of CLP($\mathcal{R}$) programs to detect dead variables. An element of the abstract domain consists of four components :

- *CA* : this set contains the so-called *call alive* variables which are those passed in from an upper level call. They may be accessed after the current call terminates.
- *SH* : possible sharing information established by the unification constraints, not by the numerical constraints (variables possibly share if the terms bound to these variables may contain a common variable). This information is similar to that of [15] for LP. A slight extension is the addition of a special element to express sharing with variables outside the scope of the current clause.
- *GR* : definiteness information, represented in the form of *Prop*-formulae [81, 84, 21].
- *NL* : information on non-linear constraints, represented in similar formulae as the *GR* information.

The *CA* component can be derived in a straightforward way. Both the *SH* and *GR* components are strongly based on previously defined abstractions in the context of LP. The *NL* component is novel (the non-linear information maintained by Hanus [48] is weaker and is not adequate for the purpose of detecting dead variables). The complete analysis shows some similarity of purpose with work of Mulkers [91, 92] for LP. The interest there is in determining dead variables and dead structures for re-use and compile-time garbage collection.

4.4 LP mode analyses

As LP is a special case of CLP, CLP mode analysis is a generalisation of LP mode analysis, where (abstract) unification is replaced by (abstract) constraint solving. Therefore, we also point out the LP mode analysis systems related to our work. These include relatively simple systems that derive freeness (and possibly also other) information, and that keep track of possible sharing between variables.

In [6], Bruynooghe presents a simple analysis for deriving freeness and groundness information (the considered modes are *ground*, *free* and *any*). In order to perform safe non-freeness propagation, the analysis keeps track of *possible sharing* or *aliasing* between pairs

of variables (note: aliasing was first studied in the context of occur-check reduction [104]). Variables possibly share if the terms to which they are bound may contain a common variable.

This mode analysis has been reformulated and extended by Musumbu [93]. Possible sharing is expressed by means of a partition of clause variables : variables belonging to a same equivalence class possibly share. Besides this information, an abstract substitution also contains a same value component. The same value information is also specified by a partition of the clause variables : variables in the same equivalence class are equal, i.e. are bound to the same term. In this approach, the possible sharing relation between variables is considered to be transitive. This does not allow a precise representation of information coming from different execution paths (i.e. resulting from different clauses defining some predicate call). Stated in another way, the abstraction cannot express that e.g. either X and Y possibly share or X and Z possibly share whereas Y and Z certainly do not share. Our kernel freeness abstraction (Chapter 5) does not suffer from this imprecision (the minimisation of Chapter 6 may lead to some loss of precision as will be explained there, although it is not as bad as applying transitivity).

Another early mode analysis including freeness information is presented by Debray [31]. The possible modes are free (f), non-free (nv), closed or ground (c), and any (d). The freeness computation is not as precise as it could be. First of all, when simulating abstract unification of two atoms p and q with abstract call substitution θ, instead of using the real atoms p and q (or their most general unifier) and θ, the abstract unification algorithm uses one of the two atoms, say q, and $p\theta$. Secondly, the sharing relation is considered to be transitive. Thirdly, sharing information is used to compute freeness information, but the effect of sharing is not considered as precisely as it could be. For example, consider $p(X, f(X)) = p(Z, W)$ with all variables being free before this unification is performed; abstract unification infers that Z and W share and, since W is non-free, also Z obtains the mode non-free. A better result could be achieved by observing that, since W is free before the unification, unifying it with $f(X)$ cannot instantiate X and hence also Z remains free. Such reasonings are performed in more recent proposals, such as [20, 95, 13, 8], which deal with sharing information in a more sophisticated way.

The cross-fertilisation between sharing and freeness derivation has been pointed out explicitly by Muthukumar and Hermenegildo in [95]. Abstract substitutions in their analysis consist of two components : a sharing component and a mode component assigning mode F (free), G (ground) or NF (potentially non-free) to each variable. The sharing information is not in the form of pair-sharing (set of variable pairs) as described above, but is expressed as so called set-sharing (first introduced by Jacobs and Langen in [53]) : the sharing component is a set of sets of variables. Intuitively, variables that occur together in a set have a variable in common. This analysis appears to capture aliasing with a higher degree of accuracy.

The integration of more precise type or structure information (cf. e.g. [59, 22]) into a freeness (or mode) analysis is not considered here. Of course, such information is indispensable in order to obtain more precise results, but it may also complicate the analysis. Also keeping track of other properties such as linearity and covering [20, 8, 9, 64] in order to increase precision, is outside the scope of our work. Our aim is primarily to develop a freeness analysis for CLP programs over several constraint domains, and to combine it with a definiteness analysis in order to obtain a first full mode analysis system for CLP.

The main issues hereby are the generalisation of unification to constraint solving and the treatment of the interaction between several constraint domains which gives rise to additional complexity. Improving precision by using a more sophisticated abstraction is a topic for further research.

4.4.1 Situating our work

In this section, we briefly summarise the main characteristics of our work, compared with the related research. This should provide a guideline for reading the following chapters.

In Section 5.1, we will introduce our notion of *freeness* that is less restrictive than the one of Marriott and Søndergaard[3] [83] : a variable is considered to be free as long as it can still take all possible values that are allowed by its type. A free variable thus constitutes a degree of freedom with respect to the satisfiability of the constraint store in which it occurs. Inferring definite constraint satisfiability is required in order to perform program optimisations such as constraint/goal reordering.

We focus mainly on the analysis of CLP(H,N) programs (extension to other constraint domains such as the PrologII tuple domain is discussed in Section 9.6). Concerning the H-part (the unification part) of a constraint, our abstraction basically corresponds to pair-sharing in LP, i.e. it keeps track of pairs of variables that may share. It also maintains which variables are possibly non-free. Concerning the N-part (numerical part) of a constraint, we abstract linear[4] numerical constraints by keeping track of just their variables and discarding the variable coefficients and constraint symbols. This contrasts with the *LSign* abstraction [87] that does maintain this more precise information. However, an important aspect that is not sufficiently elaborated in *LSign* and which we do address is the interaction between the unification and the numerical part. This leads to an abstraction that integrates the abstractions of the numerical and unification part as well as the combination or interaction of the two.

On the practical side, two orthogonal optimisations to our basic freeness analysis of Chapter 5 are presented : the first optimisation consists of retaining only minimal information that safely approximates the original freeness abstraction (Chapter 6), the second consists of combining the freeness analysis with the definiteness analysis of García de la Banda and Hermenegildo [43]. Combining the optimisations results in a practical full mode analysis system for CLP(H,N) that indicates not only which variables are free but also which are definite. To our knowledge, this is the first such system that has been developed and implemented. Results obtained with this system can be found in Chapter 11.

[3]The more restrictive freeness notion can be obtained by adding type information to our freeness analysis, as described in Section 9.5.

[4]The abstraction of non-linear numerical constraints is discussed in Section 9.3.

Chapter 5

Freeness abstraction

In this chapter we present a first abstraction for CLP(H,N) programs that keeps track of freeness information and of possible dependencies between program variables. This abstraction addresses only fundamental aspects. It forms the kernel for the more practical abstractions presented in Chapters 6, 7 and 8. In the first section, we explain the design of the abstraction and formally define the notions *freeness* and *dependency*. The second section contains the definitions of the concrete and abstract domains and of the concretisation and abstraction functions. In the third section we define the primitive abstract operations, being abstract conjunction and abstract projection. The safety conditions and some other properties of these operations are proved. Section four describes the higher-level abstract operations : abstract interpretation of a constraint, procedure-entry and procedure-exit. In order not to loose too much precision at procedure-exit, the notion of *compound abstract constraint* is introduced. Finally, we illustrate the analysis on some program examples.

5.1 Introduction

Mode information [31, 93], i.e. information on the instantiation state of variables, has proven to be very useful in the compilation of logic programs [112, 110, 108]. It is used to specialise unification. A simple form of mode analysis is a *freeness* analysis [95]. It derives for the variables at each program point whether they have mode *free* or *any*. A variable has mode *free* if it is either unbound or linked only to other free variables. It has mode *any*, if it may be bound to any term (i.e. if nothing more specific can be said about the instantiation state of the variable).
It is also well known from LP that sharing information, i.e. information on dependencies between variables, is indispensable to improve the precision of mode analysis [95, 16]. When a variable X becomes further instantiated, any free variable sharing with X may potentially become non-free. E.g. a unification $X = f(Y, Z)$ establishes sharing (or a dependency) between X and Y and between X and Z: further instantiating X potentially instantiates Y and Z and further instantiating Y or Z further instantiates X. Consequently, if sharing information is not available then all free variables become potentially non-free and all useful information is lost. To perform propagation of non-freeness in a safe way, the analysis should capture all possible dependencies between variables.

In the following sections we define a freeness and associated dependency analysis for

CLP(H,N). In that context, the freeness analysis provides information on the *"constrainedness"* state of variables rather than on their instantiation state (although these two notions coincide for non-numerical variables).

Definition 5.1.1 (freeness – non-freeness)
A variable X is free with respect to a constraint C iff $\forall v$ in the domain of X holds that $C \wedge X = v$ is consistent. Otherwise, X is non-free.

The domain of X is the set of all values that are allowed by the type of X. This general definition can be tuned towards CLP(H,N) as follows :

Definition 5.1.2 (freeness – non-freeness in CLP(H,N))
A variable X is free with respect to a constraint C iff C^* does not include a constraint $X \diamond t$ where $\diamond \in \{=, \neq\}$ and t is a non-variable Herbrand term, and C^* does not include a constraint $X \diamond n$ where $\diamond \in \{=, \neq, >, \geq\}$ and n is a number. Otherwise, X is non-free.

The analysis can easily be extended to include other constraint domains (cf. Chapter 9).

We say that C *constrains* X iff X is non-free in C. A free variable is "unconstrained" except for possible type constrainedness. This means that the variable may appear in a constraint which fixes its type, but the variable can still take all possible values allowed by its type. By definition, untyped variables are always free, whereas variables of type Herbrand are non-free; numerical variables can be either free or non-free. This notion of freeness is less restrictive than the one of Marriott and Søndergaard [83], where a variable is free only when it is unconstrained apart from possible bindings with other free variables. E.g. given a constraint $X = Y$, both definitions of freeness imply that X and Y are free; however, given a constraint $X - Y = 1$, both X and Y are non-free according to the Marriott-Søndergaard notion, whereas they are still free using Definition 5.1.2. By combining the latter freeness notion with type information, one can infer the more restrictive freeness information (cf. Section 9.5).

Free variables in the less restrictive sense constitute "degrees of freedom" with respect to the satisfiability of a constraint[1]. Satisfiability information is crucial to perform optimisations such as shifting constraints over body goals : a constraint should only be shifted over a goal (i.e. be moved later in the computation) if it cannot affect the intervening computation, i.e. if its conjunction with the constraint store in each intervening computation point is satisfiable. Besides the use for constraint shifting, freeness and dependency information also play a role in parallelisation, intelligent backtracking etc. This will be discussed in more detail in Chapter 11. We first concentrate on identifying and deriving the basic information.

The crucial operation in detecting (non-)freeness is constraint entailment. Entailed constraints that reveal non-freeness of variables can be discovered by transforming C to solved form (defined in Chapter 2), which makes these constraints explicit (follows almost immediately from the definition of the solved form). Any choice of parameters for the solved form will do.

[1] satisfiability on the value level and not on the type level

Example 5.1.1
Let $C \equiv X + Y = 6 \wedge X - Y = 2$; its solved form is $sform(C) \equiv X = 4 \wedge Y = 2$. Then X and Y are non-free.
Let $C \equiv X + Y = 6 \wedge X + Y - Z = 2 \wedge T > 0$ with $sform(C) \equiv X + Y = 6 \wedge Z = 4 \wedge T > 0$. Then Z and T are non-free whereas X and Y are free (although they are constrained to be of type numerical).
Let $C \equiv X = f(3, A), X = Y, Y = f(Z, T), Z - U + 2T = 5$. A solved form of C (with parameter A) consists of $sform(unif^*(C)) \equiv X = f(3, A) \wedge Y = f(3, A) \wedge Z = 3 \wedge T = A$ and $sform(num^*(C)) \equiv Z = 3 \wedge T - A = 0 \wedge U - 2A = -2$. Then X and Y are non-free since they are bound to a non-variable Herbrand term; also Z is non-free since C entails $Z = 3$. However, A, T and U are free (although constrained to be numerical).

Dependency information is again vital to obtain precise freeness information, as was already observed for LP. However, in the CLP(H,N) case, variable dependencies are not only established via unification constraints but also via numerical constraints or via a combination of both. This gives rise to additional complexity compared with the notion of sharing in LP.

Definition 5.1.3 (strict dependency)
A constraint C establishes a strict *dependency between a set of variables $\{X_1, \ldots, X_n\}$ with $n \geq 2$ iff $\exists X_j (1 \leq j \leq n)$ such that $[\forall i (1 \leq i \leq n, i \neq j) : \exists v_i$ in the domain of X_i such that $C \wedge x_1 = v_1 \wedge \ldots \wedge x_{j-1} = v_{j-1} \wedge x_{j+1} = v_{j+1} \wedge \ldots \wedge x_n = v_n$ constrains X_j, and there is no subset of the X_i that constrains X_j].*

In most cases, a strict dependency is symmetric, which means that "$\exists X_j$ such that" can then be replaced by "$\forall X_j$ holds that" in the above definition.

Definition 5.1.4 (dependency)
A constraint C establishes a dependency between a set of variables $\{X_1, \ldots, X_n\}$ with $n \geq 2$ iff C establishes a strict dependency between $\{X_1, \ldots, X_n\}$ or C establishes strict dependencies $S_1, \ldots, S_m$ $(m \geq 2)$ such that $\{X_1, \ldots, X_n\} = S_1 \cup \ldots \cup S_m$.

Intuitively, if a constraint C establishes a dependency between the variables $\{X_1, \ldots, X_n\}$ with $n \geq 2$, then (further) constraining all but one of the variables *may* (further) constrain the remaining variable X_j ("*may*" is used as (1) the statement does not necessarily hold for all variables X_j (unless the dependency is symmetric) and (2) only particular ways of constraining the other variables may constrain X_j). So, a dependency indicates that non-freeness or constrainedness *may* be propagated, i.e. propagation is possible but not certain. The (strict) dependency notion is illustrated by the examples below. For a numerical constraint C, a dependency $\{X_1, \ldots, X_n\}$ established by C can also be interpreted as the existence of a primitive constraint $c \in C^*$ such that $vars(c) = \{X_1, \ldots, X_n\}$. For a unification constraint C, the strict dependency information derived from C corresponds to the sharing information [6] in LP.

Example 5.1.2
Let C be the numerical constraint $X - Y - Z = 6 \wedge T - Y - Z = 3$. All variables in C are free. The equations in C directly establish the (strict) dependencies $\{X, Y, Z\}$

and $\{T, Y, Z\}$. Moreover, C entails a.o. $X - T = 3$ and $X + T - 2Y - 2Z = 9$ with corresponding dependencies $\{X, T\}$ and $\{X, T, Y, Z\}$. $\{X, T\}$ indicates that constraining X, e.g. by adding $X = 8$ to C, will cause T to be constrained, i.e. $T = 5$, and vice versa. T may also become non-free if both Y and Z are constrained as is reflected by $\{T, Y, Z\}$. The variable Y will only become non-free if both X and Z or T and Z are constrained, as indicated by the dependencies $\{X, Y, Z\}$ and $\{T, Y, Z\}$. The situation is similar for the variable Z. Note that the dependency $\{X, T, Y, Z\}$ is actually redundant (non-strict) since it does not add any new information for non-freeness propagation. In general, a union of dependencies is also a dependency but is redundant.

Let C be the unification constraint $X = f(A, B)$. Then C establishes the strict dependencies $\{X, A\}$ and $\{X, B\}$; X is non-free in C and both A and B are free. Further constraining X, e.g. by adding the primitive constraint $X = f(g(a), Y)$ will further constrain A; B however is not affected (but note that there exists another possibility for further constraining X, e.g. $X = f(Z, b)$, such that B is affected). Vice versa, constraining A (or B) will cause X to become further constrained.

Let C be the mixed constraint $X = f(Y) \wedge Y + Z = 6$. The primitive unification constraint in C establishes the (strict) dependency $\{X, Y\}$; the linear equation creates the (strict) dependency $\{Y, Z\}$. Combining the two entails the (strict) dependency $\{X, Z\}$: if X is further constrained then Z becomes non-free and, vice versa, further constraining Z will cause X to be further constrained.

A slightly more complex case is the following. Let C be the mixed constraint $X = f(Y, Z) \wedge Y + Z - T = 5$; X is non-free in C, whereas Y, Z and T are free. Further constraining X may simultaneously constrain Y and Z, thereby also constraining T; vice versa, constraining T will constrain X in the sense that it constrains the sum of the components of X (e.g. adding $T = 1$ excludes the value $f(4, 5)$ for X). So C establishes a (strict) dependency $\{X, T\}$ (besides the dependencies $\{X, Y\}, \{X, Z\}, \{Y, Z, T\}, \{X, Z, T\}$ and $\{X, Y, T\}$, where the last two are non-strict dependencies).

In order to find all strict dependencies established by a constraint C, C^* must be computed; so entailment is again the key operation. Recall that C^* is an infinite conjunction of constraints. However, it has a finite representation, i.e. the solved form of C. From this solved form, one can easily deduce all variable dependencies, although there is a complication for the numerical part. In order to find all strict dependencies established by $num^*(C)$, the easiest way is to consider all solved forms for any possible choice of parameters. For example consider $C \equiv X - 2Y - 2Z = 6 \wedge T - Y - Z = 1$. The solved form of C with parameters Y and Z is $X - 2Y - 2Z = 6 \wedge T - Y - Z = 1$; from this solved form it is not straightforward to detect the dependency between X and T. However, choosing Z and T as parameters, one obtains the solved form $Y + Z - T = -1 \wedge X - 2T = 4$ which immediately reflects the dependency between X and T.

So far, we have defined at the concrete level what kind of properties of a constraint store we want to describe, namely freeness information and the associated dependency information. In general, exact analysis of these properties may be very expensive (or even intractable), as there may be infinitely many constraint stores to be considered at some program point. An abstraction will only compute an approximation of the concrete properties. This leads to the notions *definite freeness* and *possible non-freeness* at the abstract level.

Definition 5.1.5 (definite freeness – possible non-freeness)
A variable X is said to be definitely free *at some program point if X is known to be free in any concrete constraint store that may be obtained at that point; otherwise, X is* possibly non-free *(i.e. has mode any).*

The dependency information is used to perform accurate non-freeness propagation. In order for the analysis to be safe, all *possible* dependencies between variables must be considered. The analysis computes an overestimation of the set of concrete dependencies.

As already mentioned above, the combination of constraint domains in a practical CLP language makes the analysis of CLP programs more complex, compared with the analysis of LP (Prolog) programs. One must not only abstract the constraint solving algorithm for each constraint domain, but one must also capture the interaction between the domains. In the CLP(H,N) case, the abstraction of mixed constraints cannot be separated in two independent parts – one for the primitive unification constraints and one for the primitive numerical constraints – since a program variable is often involved in both types of primitive constraints. E.g. let $C \equiv X - Y = 0 \wedge T = f(X)$; the primitive numerical constraint $X - Y = 0$ entails the primitive unification constraint $X = Y$ and influences the (abstraction of the) unification part of the constraint C (yielding a dependency between T and Y). Moreover, some dependencies span both the unification and the numerical part. E.g. $X = f(Y, Z) \wedge Y + Z - T = 5$ establishes a mixed dependency $\{X, T\}$. The abstraction that we propose in the following sections deals in an elegant way with the interaction between the two types of constraints. Propagating information from the unification part to the numerical part and vice versa is done automatically rather than via more complex explicit rules.

Furthermore, the abstraction offers a uniform representation of mode and dependency information. Dependencies are represented as sets of variables. The fact that a variable X is possibly non-free is represented by the singleton $\{X\}$. Keeping track of the information in that form allows to propagate non-freeness in an elegant way.

5.2 Concrete and abstract domain

5.2.1 Concrete domain

The elements to be abstracted are the sets of constraints that are attached to a program point in the AND-OR-graph of Bruynooghe's framework (in more general terms, the elements to be abstracted are the sets of constraints in the collecting semantics). A program point annotated with the empty set is unreachable during concrete execution. A program point annotated with $\{false\}$ implies that the last derivation step has caused failure (unsatisfiability). However, in the current abstraction, no distinction will be made between failure and unreachability due to a preceding failure or a preceding non-terminating call. So the concrete domain for the freeness abstraction, denoted Con^c, consists of sets of constraints. An element of Con^c is denoted by CS (with or without subscript) in the sequel.

Definition 5.2.1 (Concrete domain)
Let Cons be the set of constraints. The concrete domain Con^c is defined as $Con^c = \wp(Cons)$ where $\wp(S)$ denotes the powerset of the set S.

The computational order on the concrete domain is the usual order on sets.

Definition 5.2.2 (Concrete computational order)
Let $CS_1, CS_2 \in Con^{\mathcal{C}}$. *Then* $(CS_1 \leq^{\mathcal{C}} CS_2)$ *iff* $CS_1 \subseteq CS_2$.

The associated least upper bound and greatest lower bound operations are defined as follows.

Definition 5.2.3 (Concrete least upper bound)
Let $CS_1, CS_2 \in Con^{\mathcal{C}}$. *Then* $lub^{\mathcal{C}}(CS_1, CS_2) = CS_1 \cup CS_2$.

Definition 5.2.4 (Concrete greatest lower bound)
Let $CS_1, CS_2 \in Con^{\mathcal{C}}$. *Then* $glb^{\mathcal{C}}(CS_1, CS_2) = CS_1 \cap CS_2$.

These operations can easily be generalised to compute the least upper bound and greatest lower bound of more than two constraint sets. In the sequel we assume that $lub^{\mathcal{C}}$ and $glb^{\mathcal{C}}$ apply to a set of constraint sets.

Proposition 5.2.1
$(Con^{\mathcal{C}}, \leq^{\mathcal{C}}, \emptyset, Cons, lub^{\mathcal{C}}, glb^{\mathcal{C}})$ *is a complete lattice.*

PROOF
$Con^{\mathcal{C}}$ is a set with partial order $\leq^{\mathcal{C}}$; the minimal element of $Con^{\mathcal{C}}$ is $\emptyset$ and the maximal element is $Cons$. Following from Definitions 5.2.3 and 5.2.4 every subset S of $Con^{\mathcal{C}}$ has a least upper bound, being $\bigcup \{CS \mid CS \in S\}$, and a greatest lower bound, being $\bigcap \{CS \mid CS \in S\}$. □

5.2.2 Abstract domain

The abstract domain for the freeness abstraction, denoted $Con^{\mathcal{F}}$, consists of sets of sets of variables. A special minimal element $\bot$ is added to the abstract domain and denotes *definite unsatisfiability* or *unreachability* : if a program point is annotated with $\bot$ in the abstract AND-OR-graph, then the point is unreachable during concrete execution either because of failure or because of a non-terminating preceding call. An element of the abstract domain is called an *abstract constraint* and is denoted by AC (with or without subscript or superscript).

Definition 5.2.5 (Abstract domain)
Let Var be the set of program variables. The abstract domain $Con^{\mathcal{F}}$ *is defined as* $Con^{\mathcal{F}} = p(p_\emptyset(Var)) \cup \{\bot\}$ *with* $p_\emptyset(S) = p(S) \setminus \{\emptyset\}$.

The computational order $\leq^{\mathcal{F}}$ (which in this case coincides with the approximation order) on the abstract domain is defined as follows.

Definition 5.2.6 (Abstract computational order)
For every $AC_1, AC_2 \in Con^{\mathcal{F}}$ *holds :*

$$AC_1 \leq^{\mathcal{F}} AC_2 \quad \text{iff} \quad AC_1 = \bot \text{ or } (AC_1 \neq \bot, AC_2 \neq \bot \text{ and } AC_1 \subseteq AC_2).$$

The equality relation $=^{\mathcal{F}}$ based on this order is the following: $AC_1 =^{\mathcal{F}} AC_2$ iff $AC_1 \leq^{\mathcal{F}} AC_2$ and $AC_2 \leq^{\mathcal{F}} AC_1$; $AC_1 =^{\mathcal{F}} AC_2$ means that AC_1 and AC_2 both equal $\bot$ or that they are identical sets. If no confusion is possible, we write $=$ instead of $=^{\mathcal{F}}$.
The associated least upper bound and greatest lower bound operations are the following:

Definition 5.2.7 (Abstract least upper bound)
Let $AC_1, AC_2 \in Con^{\mathcal{F}}$. Then

$$lub^{\mathcal{F}}(AC_1, AC_2) = \begin{cases} AC_2 & \text{if } AC_1 = \bot \\ AC_1 & \text{if } AC_2 = \bot \\ AC_1 \cup AC_2 & \text{otherwise} \end{cases}.$$

Definition 5.2.8 (Abstract greatest lower bound)
Let $AC_1, AC_2 \in Con^{\mathcal{F}}$. Then

$$glb^{\mathcal{F}}(AC_1, AC_2) = \begin{cases} \bot & \text{if } AC_1 = \bot \text{ and/or } AC_2 = \bot \\ AC_1 \cap AC_2 & \text{otherwise} \end{cases}.$$

These operations can easily be generalised to compute the least upper bound and greatest lower bound of more than two abstract constraints. In the sequel we assume that $lub^{\mathcal{F}}$ and $glb^{\mathcal{F}}$ apply to a set of abstract constraints.

Proposition 5.2.2
$(Con^{\mathcal{F}}, \leq^{\mathcal{F}}, \bot, \wp_\emptyset(Var), lub^{\mathcal{F}}, glb^{\mathcal{F}})$ is a complete lattice.

PROOF
$Con^{\mathcal{F}}$ is a set with partial order $\leq^{\mathcal{F}}$; the minimal element of $Con^{\mathcal{F}}$ is $\bot$ and the maximal element is $\wp_\emptyset(Var)$. Following from Definitions 5.2.7 and 5.2.8, every subset S of $Con^{\mathcal{F}}$ has a least upper bound, being $\bigcup \{ AC \mid AC \in S \}$, and a greatest lower bound, being $\bigcap \{ AC \mid AC \in S \}$. □

Since the abstract domain is a complete lattice, it certainly satisfies the weaker conditions imposed by Bruynooghe's framework (cf. Section 3.2.2). Also notice that $Con^{\mathcal{F}}$ is finite since the set of program variables is finite.

5.2.3 Abstraction and concretisation function

5.2.3.1 Abstraction function

The abstraction function α maps a set of concrete constraints onto an abstract constraint.

Definition 5.2.9 (Abstraction function)
Let $CS \in Con^{\mathcal{C}}$. Then

$$\alpha(CS) = \begin{cases} \bot & \text{if } CS = \emptyset \\ lub^{\mathcal{F}}(\{ \alpha(\{C\}) \mid C \in CS \}) & \text{otherwise} \end{cases}$$

where $\alpha(\{C\})$ for $C \in Cons$ is defined below.

By abuse of notation, we usually write $\alpha(C)$ instead of $\alpha(\{C\})$.

A general definition of $\alpha(C)$ is the following :

$$\begin{aligned}\alpha(C) &= \bot \text{ if } sform(C) = false, \text{ otherwise}\\ \alpha(C) &= \{\{X\} \mid C \text{ constrains } X \text{ and } X \in vars(C)\} \cup\\ &\quad \{\{X_1,\ldots,X_n\} \mid C \text{ establishes the dependency } \{X_1,\ldots,X_n\} \subseteq vars(C)\}\end{aligned}$$

Note that for C with $sform(C) = true$ we have that $\alpha(C) = \emptyset$. The general definition is now specialised with respect to (satisfiable) CLP(H,N) constraints.

`Abstracting a numerical constraint`

First of all, we focus on the abstraction of a satisfiable numerical constraint that consists only of *linear equations*. Each primitive equation establishes a dependency between its variables. If it contains only a single variable, the variable is non-free. Therefore, abstracting an equation consists of keeping track of its variables and discarding its coefficients. To safely abstract a conjunction of equations, one must consider not only the original equations but also all entailed ones, in order to discover all non-free variables and all variable dependencies. The solved forms of C (corresponding with different choices of parameters) allow to extract all information.

Definition 5.2.10 (Abstracting a conjunction of linear equations)
Let C be a satisfiable conjunction of linear equations. Let

$$W = \left\{ \{X_1,\ldots,X_n\} \;\middle|\; \begin{array}{l}(a_1X_1+\cdots+a_nX_n = b) \in sform(C)\\ \text{for any } sform(C)\end{array} \right\}.$$

Then, $\alpha(C) = close(W)$ where $close(W)$ denotes the closure under union of W.

Taking the closure under union at the abstract level corresponds to computing all linear combinations of the primitive equations in each $sform(C)$ at the concrete level. The set of these combinations is equivalent to $lc(C)$, the conjunction of all linear combinations of the equations in C. As a consequence, the definition of $\alpha(C)$ can also be written as follows :

Definition 5.2.11 (Abstracting a conjunction of linear equations (2))
Let C be a satisfiable conjunction of linear equations. Then

$$\alpha(C) = \left\{ \{X_1,\ldots,X_n\} \;\middle|\; (a_1X_1+\cdots+a_nX_n = b) \in lc(C) \right\}.$$

Example 5.2.1
For $C \equiv X + Y = 4 \wedge Y + Z = 3$, the different solved forms (corresponding to different parameter choices) are :

$sform(C) \equiv Y + X = 4 \wedge Z - X = -1$ *(parameter X),*
$sform(C) \equiv X + Y = 4 \wedge Z + Y = 3$ *(parameter Y),*
$sform(C) \equiv X - Z = 1 \wedge Y + Z = 3$ *(parameter Z).*

So $\alpha(C) = close(\{\{X,Y\},\{Y,Z\},\{X,Z\}\}) = \{\{X,Y\},\{Y,Z\},\{X,Z\},\{X,Y,Z\}\}$. *$C$ establishes dependencies between pairs of variables; it also establishes the dependency* $\{X,Y,Z\}$ *as it entails* $X + 2*Y + Z = 7, X + 5*Y + 4*Z = 16, \ldots$ *The abstraction indicates that X, Y and Z are free and that further constraining any variable in C may cause X, Y and Z to become non-free.*
For $C \equiv X - Y - Z = 6 \wedge T - Y - Z = 3$, $\alpha(C) = close(\{\{X,Y,Z\},\{T,Y,Z\},\{X,T\}\} = \{\{X,Y,Z\},\{T,Y,Z\},\{X,T\},\{X,Y,Z,T\}\}$. *All variables are free in C. The dependency* $\{X,T\}$ *indicates that constraining X will cause T to become non-free and vice versa (note that C entails $X - T = 3$). A similar reasoning holds for the other dependencies.*

In Definition 5.2.10, the dependencies that are added via the closure operation are in fact redundant : all dependency information that is necessary for the non-freeness propagation is already present in W. However, the exhaustive enumeration of all combinations contributes to the expressive power and precision of the analysis. First of all, note that although $\alpha(C)$ is closed under union, an abstract constraint AC is not necessarily closed in general, e.g. the abstraction of a *set* of constraints may result in a non-closed abstract constraint (e.g. $\alpha(\{(X = 1),(Y = Z)\}) = \{\{X\},\{Y,Z\}\}$). Having a non-closed AC means that the union of two dependencies in AC does not necessarily belong to AC. At the concrete level, this corresponds to two primitive constraints originating from different computation paths (OR-branches) rather than being in conjunction. Secondly, when joining two constraints C_1 and C_2, more than one primitive equation in $sform(C_1)$ may be involved in deriving a primitive equation entailed by $C_1 \wedge C_2$ (and similar for $sform(C_2)$). For example, let $sform(C_1) \equiv X - A = 0 \wedge Y - B = 0$ and $sform(C_2) \equiv A + B - Z = 0$; the equation $X + Y - Z = 0$ entailed by $C_1 \wedge C_2$ is obtained by taking the linear combination $X + Y - A - B = 0$ of the primitive equations in C_1 and combining it with the equation in C_2. At the abstract level, it means that the union of dependencies in W must be computed when performing abstract conjunction. Putting these observations together leads us to the following : if closing an abstract constraint would be delayed until conjunction, then also an abstract constraint resulting from a preceding least upper bound operation would be closed. This would incur a loss of precision, since dependencies out of different computation paths would be combined. By closing $\alpha(C)$ at once there is no longer need for a closure operation at conjunction. Consequently, a larger precision is obtained. A disadvantage however is that the abstractions can become quite large. The minimal freeness abstraction described in Chapter 6 trades off exactly this form of precision for efficiency.

For the moment, we compute a safe but rough abstraction of the other numerical constraints. Linear disequations and inequalities of the form $a_1X_1 + \ldots + a_nX_n \diamond b$ with $\diamond \in \{\neq, >, \geq\}$ can be abstracted in the same way as linear equations. The worst case assumption for a satisfiable constraint of that form is that the left-hand side is equal to a single value v (i.e. $a_1X_1 + \ldots + a_nX_n = v$ with $v \neq, >$ or $\geq b$). So $C \equiv X > Z \wedge Y > Z$ is abstracted as $\alpha(C) \equiv close(\{\{X,Z\},\{Y,Z\},\{X,Y\}\})$, although there is no dependency between X and Y in the concrete case. Non-linear numerical constraints are more difficult to handle. Most of the current CLP systems delay such constraints until they become linear. A safe but imprecise way to abstract a primitive non-linear constraint c is to take the powerset of $vars(c)$ (hence considering every variable in $vars(c)$ as possibly non-free). More precise ways to handle these numerical constraints will be described in Chapter 9.

Abstracting a unification constraint

For a satisfiable unification constraint C, all variable dependencies established by C are detected by considering all possible combinations of the primitive unifications in C. An easy way to discover all dependencies consists of computing the solved form of C. This results in a conjunction of primitive unifications of the form $X = t$. Each of these establishes dependencies between X and each variable of t. Moreover, if t is a non-variable term then X is non-free, which is represented by $\{X\}$ in the abstraction. Two of the primitive unifications entail a dependency between the variables in their left-hand sides if the right-hand sides share a variable.
Recall that the solved form of a unification constraint C is isomorphic to an idempotent substitution, say θ.

Definition 5.2.12 (Abstracting a unification constraint)
Let C be a satisfiable unification constraint and θ be the substitution corresponding to $sform(C)$. Let

$$W = \left\{ \{X\} \,\middle|\, X\theta \text{ is a non-variable} \right\} \cup \left\{ \{X,Y\} \,\middle|\, X \not\equiv Y, vars(X\theta) \cap vars(Y\theta) \neq \emptyset \right\}.$$

Then, $\alpha(C) = close(W)$ where $close(W)$ denotes the closure under union of W.

As in the abstraction of a numerical constraint, taking the closure under union when computing $\alpha(C)$ contributes to the precision of the further analysis.

Example 5.2.2
Let $C_1 \equiv X = f(A,B) \wedge Y = g(A)$ and $C_2 \equiv X = f(D,E) \wedge Y = g(D) \wedge D = 3$; C_1 is in solved form and $sform(C_2) \equiv X = f(3,E) \wedge Y = g(3) \wedge D = 3$.

$$\begin{aligned} \alpha(C_1) &= close(\{\{X\},\{Y\},\{X,A\},\{X,B\},\{Y,A\},\{X,Y\}\}) \\ &= \left\{ \begin{array}{l} \{X\},\{Y\},\{X,A\},\{X,B\},\{Y,A\},\{X,Y\}, \\ \{X,A,Y\},\{X,B,Y\},\{X,A,B\},\{X,A,B,Y\} \end{array} \right\}. \end{aligned}$$

Hence, A and B are free and independent; X and A, as well as X and B, depend on each other, so further constraining X may cause A and B to become non-free and vice versa; Y and B are independent but Y and A are not. Finally, further constraining X may cause further constraining of Y and vice versa.

$$\begin{aligned} \alpha(C_2) &= close(\{\{X\},\{Y\},\{D\},\{X,E\}\}) \\ &= \left\{ \begin{array}{l} \{X\},\{Y\},\{D\},\{X,E\},\{X,D\},\{X,Y\},\{Y,D\}, \\ \{X,D,E\},\{X,Y,E\},\{X,Y,D\},\{X,Y,D,E\} \end{array} \right\}. \end{aligned}$$

Hence, E is the only free variable and there exists a dependency between E and X.

Abstracting a mixed constraint

The abstraction of a satisfiable mixed constraint C includes the abstractions of $unif^*(C)$ and $num^*(C)$. Moreover, it also contains mixed dependencies established by a combination

of some primitive constraints in $unif^*(C)$ with a primitive constraint in $num^*(C)$. Again, the solved form of C is used to compute the abstraction.
To obtain all mixed dependencies, it is sufficient to consider the primitive unification constraints in $sform(unif^*(C))$ that are of the form $X_i = \mathcal{T}^+[Y_i]$ where the right-hand side denotes a *compound* term containing the variable Y_i. Constraints of the form $X_i = Y_i$ entail a numerical constraint $X_i - Y_i = 0$ that is already taken into account in the computation of $sform(num^*(C))$. Also, one only has to consider the primitive numerical constraints in $sform(num^*(C))$ that are not of the form $Y_i - Z_i = 0$; the latter entails $Y_i = Z_i$ which is taken into account during the computation of $sform(unif^*(C))$. The combination of $X_i = \mathcal{T}^+[Y_i]$ with a relevant numerical constraint $a_1 Y_i + a'_1 Z_1 + \ldots + a'_n Z_n = b$ in $sform(num^*(C))$ establishes the dependency $\{X_i, Z_1, \ldots, Z_n\}$. Moreover, different Y_i can be replaced simultaneously : let $X_i = \mathcal{T}^+[Y_i]$ $(1 \leq i \leq m)$ be in $sform(unif^*(C))$ and $a_1 Y_1 + \ldots + a_m Y_m + a'_1 Z_1 + \ldots + a'_n Z_n = b$ in $sform(num^*(C))$; combining these primitive constraints yields the dependency $\{X_1, \ldots, X_m, Z_1, \ldots, Z_n\}$. The X_i must not necessarily be distinct (whereas the Y_i must be). Constraining a X_i may simultaneously constrain several of its components.

Definition 5.2.13 (Abstracting a mixed constraint)
Let C be a satisfiable mixed constraint, let θ be the substitution corresponding to $sform(unif^*(C))$ and let

$$
\begin{aligned}
W \;=\; & \left\{ \{X\} \;\middle|\; X\theta \text{ is a non-variable} \right\} \\
& \cup \left\{ \{X, Y\} \;\middle|\; X \not\equiv Y,\ vars(X\theta) \cap vars(Y\theta) \neq \emptyset \right\} \\
& \cup \left\{ \{X_1, \ldots, X_n\} \;\middle|\; \begin{array}{l} (a_1 X_1 + \ldots + a_n X_n = b) \in sform(num^*(C)) \\ \text{for any } sform(num^*(C)) \end{array} \right\} \\
& \cup \left\{ \begin{array}{l} \{X_1, \ldots, X_m, Z_1, \ldots, Z_n\} \;\big|\; \\ X_i\theta \text{ is a compound term, } Y_i \in vars(X_i\theta) \\ \text{all } Y_i \text{ are distinct, the } X_i \text{ are not necessarily distinct } (i = 1..m), \\ (a_1 Y_1 + \ldots + a_m Y_m + a'_1 Z_1 + \ldots + a'_n Z_n = b) \in sform(num^*(C)) \\ \text{(not of the form } Y - Z = 0) \text{ for any } sform(num^*(C)) \end{array} \right\}
\end{aligned}
$$

Then $\alpha(C) = close(W)$ where $close(W)$ denotes the closure under union of W.

The first and second set in the definition of W correspond to the abstraction of $unif^*(C)$. The third set is the abstraction of $num^*(C)$. The last set contains the mixed dependencies. Again, as in the abstraction of a numerical or unification constraint, taking the closure under union contributes to the precision of the further analysis.

Example 5.2.3
Let $C \equiv X = f(A, B) \wedge A + T = 3$. Then,

$$
\begin{aligned}
W \;&=\; \left\{ \{X\}, \{X, A\}, \{X, B\}, \{A, T\}, \{X, T\} \right\}, \\
\alpha(C) \;&=\; \left\{ \begin{array}{l} \{X\}, \{X, A\}, \{X, B\}, \{A, T\}, \{X, T\}, \{X, A, B\}, \\ \{X, A, T\}, \{X, B, T\}, \{X, A, B, T\} \end{array} \right\}.
\end{aligned}
$$

Let $C \equiv X = f(A,B) \wedge A + B + T = 3$. *Then,*

$$\begin{aligned} W &= \left\{ \{X\},\{X,A\},\{X,B\},\{A,B,T\},\{X,B,T\},\{X,A,T\},\{X,T\} \right\}, \\ \alpha(C) &= \left\{ \begin{array}{l} \{X\},\{X,A\},\{X,B\},\{A,B,T\},\{X,B,T\} \\ \{X,A,T\},\{X,T\},\{X,A,B\},\{X,A,B,T\} \end{array} \right\}. \end{aligned}$$

Let $C \equiv X = f(Y) \wedge Y = g(U) \wedge U - T - Z = 0$. *Then,*

$$\begin{aligned} W &= \left\{ \begin{array}{l} \{X\},\{Y\},\{X,Y\},\{X,U\}, \\ \{Y,U\},\{U,T,Z\},\{X,T,Z\},\{Y,T,Z\} \end{array} \right\}, \\ \alpha(C) &= \left\{ \begin{array}{l} \{X\},\{Y\},\{X,Y\},\{X,U\},\{Y,U\},\{U,T,Z\}, \\ \{X,T,Z\},\{Y,T,Z\},\{X,Y,U\},\{X,U,T,Z\}, \\ \{X,Y,T,Z\},\{Y,U,T,Z\},\{X,Y,U,T,Z\} \end{array} \right\}. \end{aligned}$$

Let $C \equiv X = f(A) \wedge U = g(B) \wedge A - 3B + T = 1$. *Then,*

$$\begin{aligned} W &= \left\{ \begin{array}{l} \{X\},\{X,A\},\{U\},\{U,B\},\{A,B,T\}, \\ \{X,B,T\},\{A,U,T\},\{X,U,T\} \end{array} \right\} \\ \alpha(C) &= close(W) \end{aligned}$$

Abstracting a projected constraint

According to our abstract interpretation framework, we must also deal with constraints that are projected onto a set of variables. The abstraction of a projected constraint is defined in terms of the abstraction of the constraint itself.

Definition 5.2.14 (Abstracting a projected constraint)
Let C *be a satisfiable constraint and let* $\exists_V C$ *be its projection onto* $V \subseteq Var$. *Then*

$$\alpha(\exists_V C) = \left\{ S \in \alpha(C) \,\middle|\, S \subseteq V \right\}$$

Mode and dependency information

The link between an abstract constraint AC and the mode and dependency information it describes is formalised in Proposition 5.2.3. The proposition follows immediately from Definitions 5.2.10, 5.2.12 and 5.2.13.

Proposition 5.2.3
Let $AC \in Con^{\mathcal{F}}$. *Then the following holds for each* $CS \in Con^{\mathcal{C}}$ *such that* $\alpha(CS) \leq^{\mathcal{F}} AC$:

1. $\{X\} \notin AC$ *implies that* X *is definitely free in each* $C \in CS$. *Alternatively,* $\{X\} \in AC$ *indicates that* X *is* possibly *non-free in some* $C \in CS$.

2. *If* $\{X_1,\ldots,X_n\} \in AC$ *then* CS possibly *establishes a dependency between the variables* $X_1,\ldots,X_n$, *i.e. (further) constraining all but one of the variables* may *(further) constrain the remaining variable. Otherwise, there is certainly no dependency* $\{X_1,\ldots,X_n\}$ *in each* $C \in CS$.

Example 5.2.4
Let $AC = \{\{X, Y, Z\}, \{T, Y, Z\}, \{X, T\}, \{X, Y, Z, T\}\}$. *Then any set of concrete constraints* CS *that is described by* AC *leaves all variables free and* possibly *establishes the dependencies* $\{X, Y, Z\}$, $\{T, Y, Z\}$ *and* $\{X, T\}$ *(*$\{X, Y, Z, T\}$ *does not yield any new dependency information with respect to the other dependencies). Examples of such a* CS *are* $CS \equiv \{X - Y - Z = 6 \wedge T - Y - Z = 3\}$ *or* $CS \equiv \{(2X + Y - Z = 4), (X + T = 5)\}$. *The abstraction of the former is exactly* AC, *whereas the latter only establishes the dependencies* $\{X, Y, Z\}$ *and* $\{X, T\}$.

5.2.3.2 Concretisation function

The concretisation function γ is uniquely determined by the abstraction function α, as described in [25].

Definition 5.2.15 (Concretisation function)
Let $AC \in Con^{\mathcal{F}}$. *Then* $\gamma(AC) = \bigcup \{\ CS \in Con^{\mathcal{C}} \mid \alpha(CS) \leq^{\mathcal{F}} AC\ \}$.

5.2.3.3 Relation between concrete and abstract domain

The relation between the concrete and abstract domain is formalised in the following proposition.

Proposition 5.2.4
$(Con^{\mathcal{C}}, \leq^{\mathcal{C}}) \underset{\alpha}{\overset{\gamma}{\rightleftarrows}} (Con^{\mathcal{F}}, \leq^{\mathcal{F}})$ *is a Galois insertion [25].*

PROOF
In order to prove that $(Con^{\mathcal{C}}, \leq^{\mathcal{C}}) \underset{\alpha}{\overset{\gamma}{\rightleftarrows}} (Con^{\mathcal{F}}, \leq^{\mathcal{F}})$ is a Galois insertion we must show that:
1. $\forall AC \in Con^{\mathcal{F}} : \alpha(\gamma(AC)) =^{\mathcal{F}} AC$; $\forall CS \in Con^{\mathcal{C}} : CS \leq^{\mathcal{C}} \gamma(\alpha(CS))$;
2. α and γ are monotonic.

The proof is as follows :

1. (a) Proof of $\forall AC \in Con^{\mathcal{F}} : \alpha(\gamma(AC)) =^{\mathcal{F}} AC$.
   Two cases can be considered :
   i. $AC =^{\mathcal{F}} \bot$. Then $\alpha(\gamma(AC)) =^{\mathcal{F}} \alpha(\emptyset) =^{\mathcal{F}} \bot$.
   ii. $AC \neq^{\mathcal{F}} \bot$. First of all, we prove that $\alpha(\gamma(AC)) \leq^{\mathcal{F}} AC$. By definition of γ, $\gamma(AC) = \bigcup \{\ CS \in Con^{\mathcal{C}} \mid \alpha(CS) \leq^{\mathcal{F}} AC\ \}$. Then $\alpha(\gamma(AC)) = \alpha(\bigcup \{\ CS \in Con^{\mathcal{C}} \mid \alpha(CS) \leq^{\mathcal{F}} AC\ \}) = \bigcup \{\ \alpha(CS) \mid CS \in Con^{\mathcal{C}}, \alpha(CS) \leq^{\mathcal{F}} AC$ *(i.e.* $\alpha(CS) \subseteq AC)$ $\}$ by Definition 5.2.9. So, $\alpha(\gamma(AC)) \subseteq AC$ or $\alpha(\gamma(AC)) \leq^{\mathcal{F}} AC$.
   Secondly, it has to be shown that $AC \leq^{\mathcal{F}} \alpha(\gamma(AC))$. Considering the definition of α, it is easy to see that $\forall AC \in Con^{\mathcal{F}} : \exists CS' \in Con^{\mathcal{C}} : \alpha(CS') =^{\mathcal{F}} AC$. It suffices to take for each $A \in AC$ a constraint C such that $\alpha(C) =^{\mathcal{F}} A$; let CS' be the set of these constraints C, then $\alpha(CS') =^{\mathcal{F}} AC$. By definition of γ, $\gamma(AC) = \bigcup \{\ CS \in Con^{\mathcal{C}} \mid \alpha(CS) \leq^{\mathcal{F}} AC\ \}$. Since $\alpha(CS') =^{\mathcal{F}} AC$, CS' is one of the CS in this formula. As mentioned above, $\alpha(\gamma(AC)) = \bigcup \{\ \alpha(CS) \mid CS \in Con^{\mathcal{C}}, \alpha(CS) \leq^{\mathcal{F}} AC\ \}$ with one of the CS being CS'. This implies that $\alpha(\gamma(AC)) \geq^{\mathcal{F}} \alpha(CS')$ or $\alpha(\gamma(AC)) \geq^{\mathcal{F}} AC$.

(b) Proof of $\forall CS \in Con^{\mathcal{C}} : CS \leq^{\mathcal{C}} \gamma(\alpha(CS))$, i.e. $CS \subseteq \gamma(\alpha(CS))$.
Two cases can be considered :

i. $CS = \emptyset$. Then $\gamma(\alpha(CS)) = \gamma(\bot) = \emptyset$.

ii. $CS \neq \emptyset$. By definition of γ, $\gamma(\alpha(CS)) = \bigcup \{ CS' \in Con^{\mathcal{C}} \mid \alpha(CS') \leq^{\mathcal{F}} \alpha(CS) \}$. So, one of the CS' is exactly CS and therefore $CS \subseteq \gamma(\alpha(CS))$ or $CS \leq^{\mathcal{C}} \gamma(\alpha(CS))$.

2. (a) α is monotonic iff $\forall CS_1, CS_2 \in Con^{\mathcal{C}} : CS_1 \leq^{\mathcal{C}} CS_2 \Rightarrow \alpha(CS_1) \leq^{\mathcal{F}} \alpha(CS_2)$.
Three cases can be considered :

i. $CS_1 = \emptyset$. Then $\alpha(CS_1) =^{\mathcal{F}} \bot$ and $\bot \leq^{\mathcal{F}} AC$ for every $AC \in Con^{\mathcal{F}}$, so also $\bot \leq^{\mathcal{F}} \alpha(CS_2)$.

ii. $CS_2 = \emptyset$. Since $CS_1 \leq^{\mathcal{C}} CS_2$, CS_1 must also be $\emptyset$. So, $\alpha(CS_1) =^{\mathcal{F}} \alpha(CS_2) =^{\mathcal{F}} \bot$ and $\bot \leq^{\mathcal{F}} \bot$.

iii. $CS_1 \neq \emptyset$ and $CS_2 \neq \emptyset$. Then it has to be proved that $CS_1 \subseteq CS_2 \Rightarrow \alpha(CS_1) \subseteq \alpha(CS_2)$. This follows immediately from the definition of α.

(b) γ is monotonic iff $\forall AC_1, AC_2 \in Con^{\mathcal{F}} : AC_1 \leq^{\mathcal{F}} AC_2 \Rightarrow \gamma(AC_1) \leq^{\mathcal{C}} \gamma(AC_2)$ (with $\leq^{\mathcal{C}}$ being $\subseteq$). Again, three cases can be considered :

i. $AC_1 =^{\mathcal{F}} \bot$. Then $\gamma(AC_1) = \emptyset$ and $\emptyset \subseteq CS$ for every $CS \in Con^{\mathcal{C}}$, so also $\emptyset \subseteq \gamma(AC_2)$.

ii. $AC_2 =^{\mathcal{F}} \bot$. Since $AC_1 \leq^{\mathcal{F}} AC_2$, AC_1 must also be $\bot$. So, $\gamma(AC_1) = \gamma(AC_2) = \emptyset$ and $\emptyset \subseteq \emptyset$.

iii. $AC_1 \neq^{\mathcal{F}} \bot$ and $AC_2 \neq^{\mathcal{F}} \bot$. Then it has to be proved that $AC_1 \subseteq AC_2 \Rightarrow \gamma(AC_1) \subseteq \gamma(AC_2)$. Consider every possible (non-empty) subset CS of $\gamma(AC_1)$. By monotonicity of α (cf. above), for each $CS \in Con^{\mathcal{C}}$ such that $CS \subseteq \gamma(AC_1)$ it must hold that $\alpha(CS) \leq^{\mathcal{F}} \alpha(\gamma(AC_1))$ (i.e. $\alpha(CS) \subseteq \alpha(\gamma(AC_1))$). By 1.(a), it is known that $\alpha(\gamma(AC_1)) = AC_1$, so $\alpha(CS) \subseteq AC_1$. Combining this with $AC_1 \subseteq AC_2$ yields $\alpha(CS) \subseteq AC_2$, which implies that $CS \subseteq \gamma(AC_2)$ (by definition of γ). Hence, $\gamma(AC_1) \subseteq \gamma(AC_2)$ or $\gamma(AC_1) \leq^{\mathcal{C}} \gamma(AC_2)$. □

5.2.4 Concrete approximation order

The abstract computational order $\leq^{\mathcal{F}}$ (which coincides with the abstract approximation order that describes the relative precision of the abstract constraints with regard to the properties of freeness and possible dependencies) induces an approximation order $\preceq^{\mathcal{C}}$ on the concrete domain.

Definition 5.2.16 (Concrete approximation order)
Let $CS_1, CS_2 \in Con^{\mathcal{C}}$. *Then* $(CS_1 \preceq^{\mathcal{C}} CS_2) \stackrel{def}{=} (\alpha(CS_1) \leq^{\mathcal{F}} \alpha(CS_2))$.

The order $\preceq^{\mathcal{C}}$ describes the relative precision of sets of constraints with respect to the concrete properties "freeness" and "variable dependencies" that they establish. The idea is that CS_1 is smaller than CS_2 if CS_1 gives rise to less variable dependencies and non-free variables. Note that this order subsumes the computational order $\leq^{\mathcal{C}}$ on $Con^{\mathcal{C}}$: if

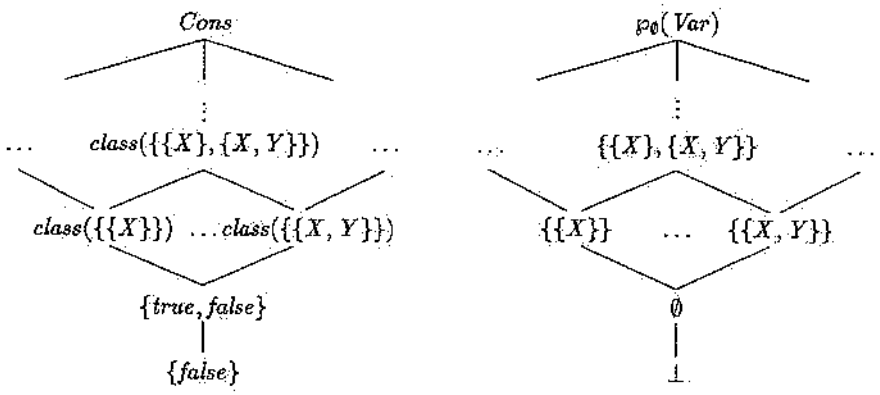


Figure 5.1: Relation between $Con^{\mathcal{C}}|_{\approx^{\mathcal{C}}}$ and $Con^{\mathcal{F}}$

$CS_1 \preceq^{\mathcal{C}} CS_2$, then also $CS_1 \preceq^{\mathcal{C}} CS_2$ (but not vice versa). For example, $\{X = 3\}$ and $\{X = 3 \wedge X + Y = 5\}$ are incomparable with respect to $\preceq^{\mathcal{C}}$, but can be compared with respect to $\preceq^{\mathcal{C}}$: $\{X = 3\} \preceq^{\mathcal{C}} \{X = 3 \wedge X + Y = 5\}$.

Let $\approx^{\mathcal{C}}$ be the equivalence relation defined as $(CS_1 \approx^{\mathcal{C}} CS_2) \stackrel{def}{=} (CS_1 \preceq^{\mathcal{C}} CS_2 \text{ and } CS_2 \preceq^{\mathcal{C}} CS_1)$. This relation divides $Con^{\mathcal{C}}$ in equivalence classes, containing sets of constraints that have the same abstraction $AC \in Con^{\mathcal{F}}$. We let $class(AC)$ denote the equivalence class of sets of constraints that are abstracted by $AC \in Con^{\mathcal{F}}$: $class(AC) = \{CS \in Con^{\mathcal{C}} \mid \alpha(CS) =^{\mathcal{F}} AC\}$. For example, $class(\{\{X, Y\}\})$ contains a.o. $\{X + Y = 3\}$ and $\{(X - Y = 5), (X + Y = 4)\}$ since both sets of constraints are abstracted by $\{\{X, Y\}\}$. Each equivalence class has a maximum element according to $\preceq^{\mathcal{C}}$ on $Con^{\mathcal{C}}$, namely the set of *all* constraints C such that $\alpha(C) \leq^{\mathcal{F}} AC$ (i.e. $\{C \in Cons \mid \alpha(C) \leq^{\mathcal{F}} AC\} = \bigcup\{CS \in Con^{\mathcal{C}} \mid \alpha(CS) \leq^{\mathcal{F}} AC\}$). This maximum element can be considered as the representative of the equivalence class. It is exactly that element that is returned by $\gamma(AC)$. All other elements of the class are subsets of the representative. Note that the representative is in general an infinite set. It is closed under entailment (at least when unsatisfiable constraints are discarded), i.e. $\forall C_1 \in \gamma(AC)$ with C_1 satisfiable: $\forall C_2$ such that $C_1 \Rightarrow C_2$ holds that $C_2 \in \gamma(AC)$. So all constraints that are weaker than $C_1 \in \gamma(AC)$ (i.e. that are entailed by C_1) also belong to $\gamma(AC)$. This notion is the opposite of the concept "closed under anti-entailment" which states that all constraints C_2 that are stronger than $C_1 \in \gamma(AC)$ (i.e. $C_2 \Rightarrow C_1$) belong to $\gamma(AC)$. The latter corresponds to the notion "substitution-closed" [32] in the context of LP.

The relation between the quotient domain $Con^{\mathcal{C}}|_{\approx^{\mathcal{C}}}$ and $Con^{\mathcal{F}}$ (which is a one-to-one correspondence) is illustrated in Figure 5.1. Note that for each $CS \in \wp(Cons)$ the set CS without unsatisfiable constraints belongs to the same equivalence class as CS itself. This means that unsatisfiable constraints are simply discarded during abstraction. How unsatisfiable constraints can be taken into consideration in order to detect definite/possible failure is discussed in Chapter 9. Some special classes in $Con^{\mathcal{C}}|_{\approx^{\mathcal{C}}}$ are designated by their representative : $\{false\}$ represents the class in which constraint sets do not contain satisfiable constraints, but only zero or more unsatisfiable constraints (all equivalent to *false*); $\{true, false\}$ represents the class of sets containing one or more trivially satisfiable con-

straints (all equivalent to *true*) and zero or more unsatisfiable constraints (all equivalent to *false*); *Cons* represents the class of all sets of constraints and provides no useful information on the variables involved (all variables are possibly non-free and all dependencies are possible).

5.3 Primitive abstract operations

5.3.1 Abstract conjunction

One of the basic operations in a CLP language is the conjunction of constraints. Recall that conjunction can be lifted to sets of constraints $CS_1, CS_2 \in Con^{\mathcal{C}}$ as follows :

$$CS_1 \wedge CS_2 = \{C_1 \wedge C_2 \mid C_1 \in CS_1, C_2 \in CS_2\}.$$

The concrete conjunction is mimicked at the abstract level by the abstract conjunction operation, denoted $AC_1 \wedge AC_2$.

Definition 5.3.1 (Abstract conjunction)
The abstract conjunction of $AC_1, AC_2 \in Con^{\mathcal{F}}$ is defined as follows :

1. *if $AC_1 = \bot$ and/or $AC_2 = \bot$, then $AC_1 \wedge AC_2 = \bot$;*
2. *otherwise*

$$\begin{array}{rcll} AC_1 \wedge AC_2 & = & AC_1 \cup AC_2 \cup (AC_1 \oplus AC_2) & \text{where} \\ AC_1 \oplus AC_2 & = & \left\{ (A_1 \cup A_2) \setminus D \;\middle|\; A_1 \in AC_1, A_2 \in AC_2, D \subseteq A_1 \cap A_2 \right\} \setminus \{\emptyset\}. & \end{array}$$

The $AC_1 \oplus AC_2$ part in the definition computes the possible dependencies that arise when combining dependencies in AC_1 with those in AC_2. Since the concrete form of the primitive constraints that established the dependencies is no longer known (the concrete coefficients of variables and the constraint symbol have been discarded), the dependencies are *possible* dependencies. In other words, a superset of the set of concrete dependencies is computed.

The conjunction operation is used to add a new constraint to the current constraint store. The effect of adding a constraint C to an abstract constraint store AC is computed by $AC \wedge \alpha(C)$. Using abstract conjunction at procedure-exit to join the abstraction of local constraints with the abstract call store guarantees the safety of the procedure-exit operation.

Example 5.3.1
Let $AC = \left\{ \{X, Y, Z\} \right\}$ and $C \equiv T - Y - Z = 3$. Then

$$\begin{array}{rcl} \alpha(C) & = & \left\{ \{T, Y, Z\} \right\} \\ AC \wedge \alpha(C) & = & \left\{ \begin{array}{l} \{X, Y, Z\}, \{T, Y, Z\}, \{X, T, Y\}, \{X, T, Z\}, \{X, T\}, \\ \{X, T, Y, Z\} \end{array} \right\} \end{array}$$

Note that AC abstracts e.g. $CS_1 = \{2X - Y - Z = 6\}$ as well as $CS_2 = \{4X + Y - Z = 5\}$; $CS_1 \wedge \{C\}$ entails $2X - T = 3$ and hence a dependency $\{X, T\}$, whereas $CS_2 \wedge \{C\}$ does not establish a dependency $\{X, T\}$ but entails dependencies $\{X, T, Y\}$ and $\{X, T, Z\}$. The results of both concrete conjunctions are captured by the abstract conjunction.

Let $AC = \left\{ \{X\} \right\}$ *and* $C \equiv X + Y - Z = 0 \wedge X + Y = 3 \wedge T = f(X)$.

$$\alpha(C) = \left\{ \begin{array}{l} \{T\},\{T,X\},\{Z\},\{X,Y,Z\},\{X,Y\},\{T,Y,Z\},\{T,Y\}, \\ \{T,Z\},\{T,X,Y,Z\},\{T,X,Y\},\{T,X,Z\} \end{array} \right\}$$

$$AC \wedge \alpha(C) = \left\{ \begin{array}{l} \{X\},\{T\},\{T,X\},\{Z\},\{X,Y,Z\},\{X,Y\}, \\ \{T,Y,Z\},\{T,Y\},\{T,Z\},\{T,X,Y,Z\}, \\ \{T,X,Y\},\{T,X,Z\},\{Y,Z\},\{Y\},\{X,Z\} \end{array} \right\}.$$

All variables are possibly non-free after conjunction.

Let $AC = \left\{ \{X,T\} \right\}$ *and* $C \equiv X + T = Z \wedge Y = f(Z)$.

$$\alpha(C) = \left\{ \{Y\},\{Y,Z\},\{X,T,Z\},\{X,T,Y\},\{X,T,Y,Z\} \right\}.$$

$$AC \wedge \alpha(C) = \left\{ \begin{array}{l} \{X,T\},\{Y\},\{Y,Z\},\{X,T,Z\},\{X,T,Y\}, \\ \{X,T,Y,Z\},\{X,Z\},\{T,Z\},\{Z\},\{X,Y\}, \\ \{T,Y\},\{X,Y,Z\},\{T,Y,Z\} \end{array} \right\}.$$

After conjunction, Z *and* Y *are possibly non-free, while* T *and* X *are definitely free.*

5.3.2 Abstract projection

The other primitive operation is projection. Projection restricts attention to a subset of the variables in a constraint. Recall that the concrete projection of a set of constraints $CS \in Con^{\mathcal{C}}$ onto a set of variables $V \subseteq Var$ is defined as :

$$\overline{\exists}_V CS = \{\overline{\exists}_V C \mid C \in CS\}.$$

Its abstract counterpart is the abstract projection operation, denoted $\overline{\exists}_V AC$.

Definition 5.3.2 (Abstract projection)
The abstract projection of $AC \in Con^{\mathcal{T}}$ *on* $V \subseteq Var$ *is defined as follows :*

1. *if* $AC = \bot$, *then* $\overline{\exists}_V AC = \bot$;
2. *otherwise* $\overline{\exists}_V AC = \{S \in AC \mid S \subseteq V\}$.

Example 5.3.2
Let $AC = \left\{ \{X,Y,Z\},\{T\},\{T,Z\},\{T,X,Y\} \right\}$; *AC abstracts for example* $CS \equiv \{(X - Y + Z = 3 \wedge T = g(Z)), (T - 3X - Y = 4)\}$. *Then* $\overline{\exists}_{\{X,Y,T\}} AC = \left\{ \{T\},\{T,X,Y\} \right\}$ *which correctly abstracts* $\overline{\exists}_{\{X,Y,T\}} CS$.

5.3.3 Properties of conjunction

This section shows a number of interesting properties of the abstract conjunction and the related $\oplus$ operation, which are used in the remainder of this chapter. The main part of the section consists of proving the safety of abstract conjunction.

5.3.3.1 General properties

Proposition 5.3.1 (Distributivity of $\oplus$ w.r.t. $\cup$)
Let $AC_1, AC_2, AC_3 \in Con^{\mathcal{F}} \setminus \{\bot\}$.
Then $(AC_1 \cup AC_2) \oplus AC_3 = (AC_1 \oplus AC_3) \cup (AC_2 \oplus AC_3)$.

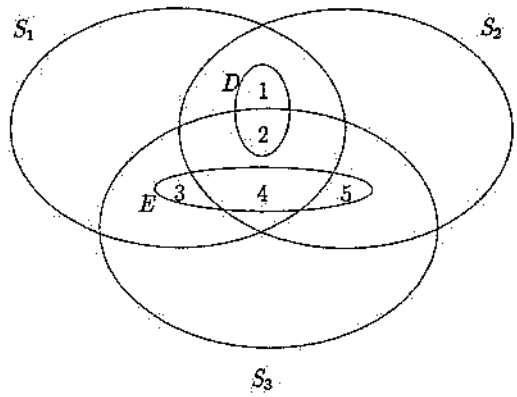
PROOF
$(AC_1 \cup AC_2) \oplus AC_3$
$= \{(B_1 \cup B_2) \setminus D \mid B_1 \in AC_1 \cup AC_2, B_2 \in AC_3, D \subseteq B_1 \cap B_2\} \setminus \{\emptyset\}$ (by definition of $\oplus$)
$= \{(B_1 \cup B_2) \setminus D \mid B_1 \in AC_1, B_2 \in AC_3, D \subseteq B_1 \cap B_2\} \setminus \{\emptyset\}$
$\quad \cup \{(B_1 \cup B_2) \setminus D \mid B_1 \in AC_2, B_2 \in AC_3, D \subseteq B_1 \cap B_2\} \setminus \{\emptyset\}$
$\quad$ (by case splitting : $B_1 \in AC_1 \cup AC_2 \Rightarrow B_1 \in AC_1$ or $B_1 \in AC_2$)
$= (AC_1 \oplus AC_3) \cup (AC_2 \oplus AC_3)$ (by definition of $\oplus$). □

Proposition 5.3.2 (Distributivity of Λ w.r.t. $\cup$)
Let $AC_1, AC_2, AC_3 \in Con^{\mathcal{F}} \setminus \{\bot\}$.
Then $(AC_1 \cup AC_2) \Lambda AC_3 = (AC_1 \Lambda AC_3) \cup (AC_2 \Lambda AC_3)$.

PROOF
$(AC_1 \cup AC_2) \Lambda AC_3$
$= (AC_1 \cup AC_2) \cup AC_3 \cup ((AC_1 \cup AC_2) \oplus AC_3)$ (by Definition 5.3.1 of Λ)
$= (AC_1 \cup AC_2) \cup AC_3 \cup ((AC_1 \oplus AC_3) \cup (AC_2 \oplus AC_3))$ (by distributivity of $\oplus$ w.r.t. $\cup$ (Proposition 5.3.1))
$= (AC_1 \cup AC_3 \cup (AC_1 \oplus AC_3)) \cup (AC_2 \cup AC_3 \cup (AC_2 \oplus AC_3))$ (by commutativity, associativity and idempotence of $\cup$)
$= (AC_1 \Lambda AC_3) \cup (AC_2 \Lambda AC_3)$ (by Definition 5.3.1 of Λ). □

Proposition 5.3.3 (Associativity of $\oplus$)
Let $AC_1, AC_2, AC_3 \in Con^{\mathcal{F}} \setminus \{\bot\}$. *Then* $(AC_1 \oplus AC_2) \oplus AC_3 = AC_1 \oplus (AC_2 \oplus AC_3)$.

PROOF
For each $A \in (AC_1 \oplus AC_2) \oplus AC_3$ it must be shown that $A \in AC_1 \oplus (AC_2 \oplus AC_3)$, and vice versa.
Let $A \in (AC_1 \oplus AC_2) \oplus AC_3$. By definition of $\oplus$, we have : $A = ([(S_1 \cup S_2) \setminus D] \cup S_3) \setminus E$ with $S_1 \in AC_1$, $S_2 \in AC_2$, $S_3 \in AC_3$, $D \subseteq S_1 \cap S_2$ and $E \subseteq [(S_1 \cup S_3) \setminus D] \cap S_3$.
This can be rewritten as (using the property $(V \setminus T) \cup W = (V \cup W) \setminus (T \setminus W)$) :
$A = [(S_1 \cup S_2 \cup S_3) \setminus D_1] \setminus E = (S_1 \cup S_2 \cup S_3) \setminus (D_1 \cup E)$ with $D_1 = D \setminus S_3$.
To show that A also belongs to $AC_1 \oplus (AC_2 \oplus AC_3)$, one must be able to write A as $(S_1 \cup [(S_2 \cup S_3) \setminus F]) \setminus G$ with $F \subseteq S_2 \cap S_3$ and $G \subseteq [(S_2 \cup S_3) \setminus F] \cap S_1$. This is achieved by defining F and G as follows (F and G are constructed from D and E parts) : $F = D_2 \cup (E \setminus S_1)$ with $D_2 = D \setminus D_1 = D \cap S_3$ and $G = D_1 \cup (E \setminus (E \setminus S_1)) = D_1 \cup (E \cap S_1)$.
The following figure illustrates the situation :
The sets correspond to the following areas: D is $D_1 \cup D_2$ with D_1 corresponding to 1 and D_2 to 2, E is $3 \cup 4 \cup 5$, F is $2 \cup 5$, G is $1 \cup 3 \cup 4$, A corresponds to the complete area $S_1 \cup S_2 \cup S_3$ except for $1 \cup 3 \cup 4 \cup 5$.



Note that $F \setminus S_1 = (D_2 \setminus S_1) \cup (E \setminus S_1) = E \setminus S_1$ since $D_2 \setminus S_1 = \emptyset$ ($D_2 \subseteq D \subseteq (S_1 \cap S_2)$); hence $(F \setminus S_1) \cup G = (E \setminus S_1) \cup D_1 \cup (E \setminus (E \setminus S_1)) = E \cup D_1$. This shows that the values for F and G are correct since $A = (S_1 \cup [(S_2 \cup S_3) \setminus F]) \setminus G = (S_1 \cup S_2 \cup S_3) \setminus ((F \setminus S_1) \cup G)$ $= (S_1 \cup S_2 \cup S_3) \setminus (D_1 \cup E)$, where the latter is exactly the form of A given above.

It still has to be shown that $F \subseteq S_2 \cap S_3$ and $G \subseteq [(S_2 \cup S_3) \setminus F] \cap S_1$.

proof of $F \subseteq S_2 \cap S_3$
$F = D_2 \cup (E \setminus S_1)$. First of all, $D_2 \subseteq S_2 \cap S_3$ since $D_2 = D \cap S_3$ and $D \subseteq S_1 \cap S_2$. Secondly, $E \setminus S_1 \subseteq S_2 \cap S_3$ since $E \subseteq [(S_1 \cup S_2) \setminus D] \cap S_3 \subseteq (S_1 \cup S_2) \cap S_3$. Combining these yields $F \subseteq S_2 \cap S_3$.

proof of $G \subseteq [(S_2 \cup S_3) \setminus F] \cap S_1$
$G = D_1 \cup (E \cap S_1)$. We show that $D_1 \subseteq [(S_2 \cup S_3) \setminus F] \cap S_1$ and $E \cap S_1 \subseteq [(S_2 \cup S_3) \setminus F] \cap S_1$.

1. Since $D_1 = D \setminus S_3$, proving that $D_1 \subseteq [(S_2 \cup S_3) \setminus F] \cap S_1$ comes down to proving that $D_1 \subseteq (S_2 \setminus F) \cap S_1$. We show that $D_1 \cap F = \emptyset$ (recall : $F = D_2 \cup (E \setminus S_1)$) :
   - $D_2 = D \setminus D_1$ implies that $D_1 \cap D_2 = \emptyset$.
   - $E \subseteq [(S_1 \cup S_2) \setminus D] \cap S_3$ (so $E \cap D = \emptyset$) and $D_1 \subseteq D$ imply that $D_1 \cap E = \emptyset$ and hence $D_1 \cap (E \setminus S_1) = \emptyset$.

   Since $D_1 \cap F = \emptyset$, $D_1 \subseteq (S_2 \setminus F) \cap S_1$ iff $D_1 \subseteq S_2 \cap S_1$. The latter is satisfied since $D_1 \subseteq D \subseteq S_1 \cap S_2$. The above allows to conclude that $D_1 \subseteq [(S_2 \cup S_3) \setminus F] \cap S_1$.
2. We first show that $(E \cap S_1) \cap F = \emptyset$ (recall : $F = D_2 \cup (E \setminus S_1)$) :
   - $(E \cap S_1) \cap D_2 = \emptyset$ since $E \cap D = \emptyset$ and $D_2 \subseteq D$.
   - $(E \cap S_1) \cap (E \setminus S_1) = \emptyset$.

   So, proving that $(E \cap S_1) \subseteq [(S_2 \cup S_3) \setminus F] \cap S_1$ comes down to proving that $(E \cap S_1) \subseteq (S_2 \cup S_3) \cap S_1$. By definition, $E \subseteq [(S_1 \cup S_2) \setminus D] \cap S_3$, so $E \subseteq S_3 \subseteq S_2 \cup S_3$. This implies that $E \cap S_1 \subseteq (S_2 \cup S_3) \cap S_1$. To conclude, $E \cap S_1 \subseteq [(S_2 \cup S_3) \setminus F] \cap S_1$.

Given 1 and 2 and the definition of G, we obtain that $G \subseteq [(S_2 \cup S_3) \setminus F] \cap S_1$.

So far, we proved that each A in $(AC_1 \oplus AC_2) \oplus AC_3$ also belongs to $AC_1 \oplus (AC_2 \oplus AC_3)$. Proving the other direction is completely analogous. □

Proposition 5.3.4 (Associativity of Λ)
Let $AC_1, AC_2, AC_3 \in Con^{\mathcal{F}}$. Then $(AC_1 \Lambda AC_2) \Lambda AC_3 = AC_1 \Lambda (AC_2 \Lambda AC_3)$.

PROOF
The proof is trivial if one or more of the AC_i $(1 \leq i \leq 3)$ equal $\bot$. Otherwise the proof is as follows.

$$
\begin{aligned}
& (AC_1 \Lambda AC_2) \Lambda AC_3 \\
= \; & (AC_1 \Lambda AC_2) \cup AC_3 \cup [(AC_1 \Lambda AC_2) \oplus AC_3] \text{ (by Definition 5.3.1 of } \Lambda) \\
= \; & [AC_1 \cup AC_2 \cup (AC_1 \oplus AC_2)] \cup AC_3 \cup [(AC_1 \cup AC_2 \cup (AC_1 \oplus AC_2)) \oplus AC_3] \\
& \text{(by Definition 5.3.1 of } \Lambda) \\
= \; & [AC_1 \cup AC_2 \cup (AC_1 \oplus AC_2)] \cup AC_3 \cup [(AC_1 \oplus AC_3) \cup (AC_2 \oplus AC_3) \cup \\
& ((AC_1 \oplus AC_2) \oplus AC_3)] \text{ (by distributivity of } \oplus \text{ w.r.t. } \cup \text{ (Proposition 5.3.1))} \\
= \; & [AC_1 \cup AC_2 \cup (AC_1 \oplus AC_2)] \cup AC_3 \cup [(AC_1 \oplus AC_3) \cup (AC_2 \oplus AC_3) \cup \\
& (AC_1 \oplus (AC_2 \oplus AC_3))] \text{ (by associativity of } \oplus \text{ (Proposition 5.3.3))} \\
= \; & AC_1 \cup [AC_2 \cup AC_3 \cup (AC_2 \oplus AC_3)] \cup [AC_1 \oplus (AC_2 \cup AC_3 \cup (AC_2 \oplus AC_3))] \\
& \text{(by commutativity and associativity of } \cup \text{ and distributivity of } \oplus \text{ w.r.t. } \cup) \\
= \; & AC_1 \cup (AC_2 \Lambda AC_3) \cup [AC_1 \oplus (AC_2 \Lambda AC_3)] \text{ (by Definition 5.3.1 of } \Lambda) \\
= \; & AC_1 \Lambda (AC_2 \Lambda AC_3) \text{ (by Definition 5.3.1 of } \Lambda). \qquad \square
\end{aligned}
$$

When abstracting a constraint, closure under union is performed (cf. Definitions 5.2.10, 5.2.12 and 5.2.13). We now formally define a *closed abstract constraint*.

Definition 5.3.3 (Closed abstract constraint)
$AC \in Con^{\mathcal{F}} \setminus \{\bot\}$ is closed (under union) iff for each two sets A_1, $A_2 \in AC$: $A_1 \cup A_2 \in AC$.

Proposition 5.3.5 ($\oplus$ preserves closedness)
If $AC_1, AC_2 \in Con^{\mathcal{F}} \setminus \{\bot\}$ are two closed abstract constraints, then $AC_1 \oplus AC_2$ is also closed.

PROOF
For each two sets A_1 and A_2 in $AC_1 \oplus AC_2$ it must be shown that $(A_1 \cup A_2) \in (AC_1 \oplus AC_2)$.
Using the definition of $\oplus$, A_1 and A_2 can be written as follows :
$A_1 = (R_1 \cup R_2) \setminus E$ with $R_1 \in AC_1$, $R_2 \in AC_2$ and $E \subseteq (R_1 \cap R_2)$,
$A_2 = (S_1 \cup S_2) \setminus F$ with $S_1 \in AC_1$, $S_2 \in AC_2$ and $F \subseteq (S_1 \cap S_2)$.
Then $A_1 \cup A_2 = ((R_1 \cup R_2) \setminus E) \cup ((S_1 \cup S_2) \setminus F)$ with $E \subseteq (R_1 \cap R_2)$ and $F \subseteq (S_1 \cap S_2)$.
Based on the property $(A \setminus E) \cup (B \setminus F) = (A \cup B) \setminus T$ where $T = (E \setminus (B \setminus F)) \cup (F \setminus (A \setminus E))$, it can be rewritten as $A_1 \cup A_2 = ((R_1 \cup S_1) \cup (R_2 \cup S_2)) \setminus T$

$$
\begin{aligned}
\text{with } T \; & = (E \setminus ((S_1 \cup S_2) \setminus F)) \cup (F \setminus ((R_1 \cup R_2) \setminus E)) \\
& \subseteq (E \cup F) \text{ (using the property } (E \setminus A) \cup (F \setminus B) \subseteq E \cup F \text{)} \\
& \subseteq ((R_1 \cap R_2) \cup (S_1 \cap S_2)) = (R_1 \cup S_1) \cap (R_1 \cup S_2) \cap (R_2 \cup S_1) \cap (R_2 \cup S_2) \\
& \qquad \text{(by applying distributivity of } \cup \text{ with respect to } \cap) \\
& \subseteq (R_1 \cup S_1) \cap (R_2 \cup S_2).
\end{aligned}
$$

So, $(A_1 \cup A_2) \in (AC_1 \oplus AC_2)$ by definition of $\oplus$ ($A_1 \cup A_2 = (B_1 \cup B_2) \setminus D$ with $B_1 = (R_1 \cup S_1) \in AC_1$ (since R_1 and $S_1 \in AC_1$ and AC_1 is closed), $B_2 = (R_2 \cup S_2) \in AC_2$ (since $R_2, S_2 \in AC_2$ and AC_2 is closed) and $D = T \subseteq (B_1 \cap B_2)$). $\square$

Proposition 5.3.6 ($\oplus$ preserves order)
Let $AC_1, AC_2, AC_3 \in Con^{\mathcal{F}} \setminus \{\bot\}$. If $AC_1 \subseteq AC_2$, then $AC_1 \oplus AC_3 \subseteq AC_2 \oplus AC_3$.

PROOF
$AC_1 \oplus AC_3$
$= \{(B_1 \cup B_3) \setminus D \mid D \subseteq B_1 \cap B_3, B_1 \in AC_1, B_3 \in AC_3\} \setminus \{\emptyset\}$ (by definition of $\oplus$)
$\subseteq \{(B_2 \cup B_3) \setminus D \mid D \subseteq B_2 \cap B_3, B_2 \in AC_2, B_3 \in AC_3\} \setminus \{\emptyset\}$ (since $AC_1 \subseteq AC_2$)
$= AC_2 \oplus AC_3$ (by definition of $\oplus$). □

The proofs of Propositions 5.3.7 and 5.3.8 below are based on Definition 5.3.1 of abstract conjunction. Considering this definition, there are three possible ways to obtain an $A \in AC_1 \wedge AC_2$:
(a1) $A \in AC_1$
(a2) $A \in AC_2$
(a3) $A \in (AC_1 \oplus AC_2)$, i.e. there exists a $B_1 \in AC_1$ and a $B_2 \in AC_2$ such that $A = (B_1 \cup B_2) \setminus D$ with $D \subseteq (B_1 \cap B_2)$.

Proposition 5.3.7 (Abstract conjunction preserves closedness)
If $AC_1, AC_2 \in Con^{\mathcal{F}} \setminus \{\bot\}$ are two closed abstract constraints, then $AC_1 \wedge AC_2$ is also closed.

PROOF
For each two sets A_1 and A_2 in $AC_1 \wedge AC_2$ it must be shown that $(A_1 \cup A_2) \in (AC_1 \wedge AC_2)$. As mentioned above, there are three possible ways to obtain A_1 as well as A_2 :
1. $A_1 \in AC_1$, 2. $A_1 \in AC_2$, 3. $A_1 \in (AC_1 \oplus AC_2)$
and a. $A_2 \in AC_1$, b. $A_2 \in AC_2$, c. $A_2 \in (AC_1 \oplus AC_2)$.
This leads to 9 cases for $A_1 \cup A_2$. In each of these cases, $A_1 \cup A_2$ can be written as an element of $AC_1 \wedge AC_2$ (i.e. in one of the three ways mentioned above) :

1a. $A_1 \in AC_1, A_2 \in AC_1$:
since AC_1 is closed, $(A_1 \cup A_2) \in AC_1$; so $(A_1 \cup A_2) \in (AC_1 \wedge AC_2)$ by case (a1)

2b. $A_1 \in AC_2, A_2 \in AC_2$:
analogous to case (1a)

1b. $A_1 \in AC_1, A_2 \in AC_2$:
$(A_1 \cup A_2) \in (AC_1 \wedge AC_2)$ by case (a3) with $B_1 = A_1$, $B_2 = A_2$ and $D = \emptyset$

2a. analogous to case (1b)

1c. $A_1 \in AC_1, A_2 = (R_1 \cup R_2) \setminus E$ with $R_1 \in AC_1$, $R_2 \in AC_2$ and $E \subseteq (R_1 \cap R_2)$:
$A_1 \cup A_2 = A_1 \cup ((R_1 \cup R_2) \setminus E)$
$= ((A_1 \cup R_1) \cup R_2) \setminus E'$ with $E' = E \setminus A_1$
(using the property $A \cup (B \setminus C) = (A \cup B) \setminus (C \setminus A)$)
where $E' \subseteq (R_1 \cap R_2) \setminus A_1 \subseteq (R_1 \cap R_2) \subseteq ((A_1 \cup R_1) \cap R_2)$.
So, $(A_1 \cup A_2) \in (AC_1 \wedge AC_2)$ by case (a3) with $B_1 = (A_1 \cup R_1) \in AC_1$ (since A_1 and $R_1 \in AC_1$ and AC_1 is closed), $B_2 = R_2 \in AC_2$ and $D = E' \subseteq (B_1 \cap B_2)$.

2c. $A_1 \in AC_2, A_2 = (R_1 \cup R_2) \setminus E$ with $R_1 \in AC_1$, $R_2 \in AC_2$ and $E \subseteq (R_1 \cap R_2)$:
analogous to case (1c)

3a. $A_1 = (R_1 \cup R_2) \setminus E$ with $R_1 \in AC_1$, $R_2 \in AC_2$ and $E \subseteq (R_1 \cap R_2)$, $A_2 \in AC_1$:
analogous to case (1c)

3b. $A_1 = (R_1 \cup R_2) \setminus E$ with $R_1 \in AC_1$, $R_2 \in AC_2$ and $E \subseteq (R_1 \cap R_2)$, $A_2 \in AC_2$:
analogous to case (1c)

3c. $A_1 \in (AC_1 \ominus AC_2)$, $A_2 \in (AC_1 \oplus AC_2)$:
$A_1 \cup A_2 \in (AC_1 \oplus AC_2)$ by Proposition 5.3.5 ($\oplus$ preserves closedness).
Hence $A_1 \cup A_2 \in (AC_1 \wedge AC_2)$ by case (a3).

□

Proposition 5.3.8 (Abstract conjunction preserves order)
Let $AC_1, AC_1', AC_2, AC_2' \in Con^{\mathcal{F}}$. If $AC_1 \leq^{\mathcal{F}} AC_1'$ and $AC_2 \leq^{\mathcal{F}} AC_2'$, then $(AC_1 \wedge AC_2) \leq^{\mathcal{F}} (AC_1' \wedge AC_2')$.

PROOF
If $AC_1 = \bot$ and/or $AC_2 = \bot$, then $AC_1 \wedge AC_2 = \bot$; since $\bot$ is the minimal element of $Con^{\mathcal{F}}$ it holds that $(AC_1 \wedge AC_2) \leq^{\mathcal{F}} (AC_1' \wedge AC_2')$. Also, if $AC_1' = \bot$, then $AC_1 = \bot$ since $AC_1 \leq^{\mathcal{F}} AC_1'$ and $\bot$ is the minimal element; so $AC_1 \wedge AC_2$ and $AC_1' \wedge AC_2'$ both equal $\bot$ and therefore $(AC_1 \wedge AC_2) \leq^{\mathcal{F}} (AC_1' \wedge AC_2')$. A similar reasoning can be applied if $AC_2' = \bot$.

Now assume that AC_1, AC_2, AC_1' and AC_2' are all different from $\bot$. Then $\leq^{\mathcal{F}}$ means $\subseteq$. For every $A \in (AC_1 \wedge AC_2)$, it has to be shown that $A \in (AC_1' \wedge AC_2')$. Three cases have to be considered, corresponding to the three possible ways in which $A \in (AC_1 \wedge AC_2)$ can be obtained :

1. $A \in AC_1$ (case (a1))
   $\Rightarrow A \in AC_1'$ since $AC_1 \subseteq AC_1'$
   $\Rightarrow A \in (AC_1' \wedge AC_2')$ by case (a1).

2. $A \in AC_2$ (case (a2))
   analogous to (1).

3. $A \in (AC_1 \oplus AC_2)$, i.e. there exists a $B_1 \in AC_1$ and a $B_2 \in AC_2$ such that $A = (B_1 \cup B_2) \setminus D$ with $D \subseteq (B_1 \cap B_2)$ (case (a3)).
   $B_1 \in AC_1 \Rightarrow B_1 \in AC_1'$ since $AC_1 \subseteq AC_1'$; $B_2 \in AC_2 \Rightarrow B_2 \in AC_2'$ since $AC_2 \subseteq AC_2'$.
   So, $A = (B_1 \cup B_2) \setminus D$ with $D \subseteq (B_1 \cap B_2)$ and $B_1 \in AC_1'$ and $B_2 \in AC_2'$
   $\Rightarrow A \in AC_1' \oplus AC_2' \Rightarrow A \in (AC_1' \wedge AC_2')$ by case (a3).

□

5.3.3.2 Safety of abstract conjunction

First of all, some concepts and notations are introduced. We use $\mathcal{T}[r]$ to denote a term in $T(\Sigma_H, Var)$ containing the subterm r, $\mathcal{T}^+[r]$ denotes a *compound* term in $T(\Sigma_H, Var)$ containing r. Syntactic identity of terms is denoted by $\equiv$, whereas $=$ denotes equality. A *selector* s in a term t is a finite sequence of integers used to indicate the position of a subterm of t. The empty selector is denoted by ε and "." denotes the concatenation of selectors. The notation t/s denotes the subterm of t selected by s, e.g. $f(g(U), V, h(Y, B))/3.1 = Y$.

Definition 5.3.4 (Selector and selected subterm)
Let t be a term. The set of selectors in t is defined as

$$Sel(t) \;=\; \{\varepsilon\} \cup \left\{ i.s \;\middle|\; \begin{array}{l} t \equiv f(t_1, \ldots, t_n) \text{ for some functor } f \text{ of arity } n, \\ 1 \leq i \leq n \text{ and } s \in Sel(t_i) \end{array} \right\}.$$

Let $s \in Sel(t)$. Then t/s is defined as

$$t/s \;=\; \text{if } (s = \varepsilon) \text{ then } t \text{ else } t_i/s', \text{ where } s = i.s',\ t \equiv f(t_1, \ldots, t_n) \text{ and } 1 \leq i \leq n.$$

Definition 5.3.5 (Compatible selectors)
The selectors s_1 and s_2 in two unifiable terms t_1 and t_2 are called compatible *if one is a subsequence of the other.*

Let $r_1 = t_1/s_1$ and $r_2 = t_2/s_2$; if s_1 and s_2 are compatible selectors in t_1 and t_2, then the equation $t_1 = t_2$ entails either $r_1 = \mathcal{T}[r_2]$ or $r_2 = \mathcal{T}[r_1]$ (note : $r_1 = r_2$ is a special case of these).
To simplify the presentation in the rest of this section, we assume that primitive unification constraints are written in *normal form* $X = t$, where the left-hand side is a variable and the (possibly) more complex term t is on the right-hand side. This does not infer a loss of generality, since each primitive constraint can be put into that form : $t = X$ can easily be transformed to $X = t$, and $t_1 = t_2$ where t_1 and t_2 are compound terms can be peeled down to $X_1 = r_1 \wedge \ldots \wedge X_n = r_n$. E.g. $f(X, g(Y)) = f(A, B)$ reduces to $X = A \wedge B = g(Y)$. Notice that the abstraction of such a $X_i = r_i$ is included in the abstraction of $t_1 = t_2$. By definition, the primitive constraints in the solved form of a unification constraint are in normal form.

Numerical terms, i.e. terms in $T(\Sigma_N, Var)$, are denoted by e and d (with or without subscripts); a and b (with or without subscripts) denote numbers. A numerical term is written in *normal form* as $a_1 * X_1 + \ldots + a_m * X_m + b$ where the X_i are distinct variables and the a_i are non-zero numbers ($1 \leq i \leq m$). If no confusion is possible, the $*$ operator is not written out explicitly.
The symbol $\dot{+}$ is used to denote the addition of numerical terms in normal form which may share variables, e.g. $(2X + 3Y) \dot{+} (X - 5Z) \dot{+} (Y - 2)$. The meaning of $\dot{+}$ is identical to the meaning of the usual $+$ operator. An expression containing the special $\dot{+}$ symbol can easily be transformed (via symbolic computation) into a numerical term in normal form, e.g. $(2X + 3Y) \dot{+} (X - 5Z) \dot{+} (Y - 2)$ is normalised into $3X + 4Y - 5Z - 2$. By $\doteq$, we denote equivalence after normalisation, e.g. $3X + 4Y - 5Z - 2 \doteq (2X + 3Y) \dot{+} (X - 5Z) \dot{+} (Y - 2)$.

Concerning primitive numerical constraints, attention is focussed on linear equations. An equation is assumed to be in the *normal form*[2] $a_1X_1 + \ldots + a_mX_m = b$ where the a_i are non-zero numbers ($1 \leq i \leq m$) and the leftmost coefficient a_1 is 1. Notice that primitive constraints in the solved form of a constraint are by definition in normal form. When we want to emphasize a subexpression e_1, the equation will be written as $e_1 = e_2$ (e.g. $X+3Y-2Z+T=1$ can be written as $3Y-2Z=1-X-T$ to emphasize the occurrence of $3Y-2Z$). Finally, a constraint containing $\dot{+}$ terms can easily be transformed into normal form, e.g. $(2X+3Y) \dot{+} (X-5Z) = 1$ is normalised into $X + Y - 5/3Z = 1/3$.

The abstraction of a constraint C is defined in terms of its solved form(s). To prove the safety of abstract conjunction with respect to concrete conjunction, it is important to know how a primitive constraint c in some solved form of C is derived from the primitive constraints in C. A number of *derivation rules* are extracted from the solved form algorithms described in Section 2.2. The fundamental operation in these algorithms is variable elimination, which results from the combination of primitive constraints.

For the unification part, a simplified version of the Martelli-Montanari algorithm (given in section 2.2) can be used since constraints are assumed to be in normal form. The possible steps in the transformation to solved form are then :

1. substitute and peel when necessary (subsumes steps 1 and 4 of Algorithm 2.2.1) :
   $X = t \wedge C_{rest}$ where $t \not\equiv X$ and $X \in vars(C_{rest}) \rightarrow$ if $X \in vars(t)$ then return false; otherwise transform to $X = t \wedge C_{rest}[X \leftarrow t \;\&\; peel]$, i.e. substitute X by t in every equation of C_{rest} and peel when necessary. Two cases can be distinguished :

   (a) substitution of $X = t_1$ in $X = t_2$ leads to $Y = t_3$ for each Y and t_3 such that ($Y = t_1/s_1$ and $t_3 = t_2/s_1$) or ($t_3 = t_1/s_1$ and $Y = t_2/s_1$);

   (b) substitution of $X = t$ in $Y = \mathcal{T}_1^+[X]$[3] leads to $Y = \mathcal{T}_2^+[t]$ with $X = \mathcal{T}_1^+[X]/s_1$ and $t = \mathcal{T}_2^+[t]/s_1$.

2. remove (corresponds to step 2 in Algorithm 2.2.1) :
   $X = X \wedge C_{rest} \rightarrow$ transform to C_{rest}

Each step in the simplified algorithm ensures that inferred constraints are in normal form, so step 3 of Algorithm 2.2.1 (i.e. $t = X \wedge C_{rest} \rightarrow X = t \wedge C_{rest}$) is no longer needed.
Two derivation rules are extracted from the simplified algorithm. They show how a primitive constraint is obtained from two other constraints during the transformation to solved form and correspond to Cases (1a) and (1b) above (Case (2) is irrelevant since it does not contribute to a constraint in the solved form). In the rules, emphasis is put on some variable in the right-hand side of each equation[4].

[2] A primitive constraint can be put into normal form by multiplying it by the appropriate non-zero number. For example, multiplying $2X - 6Y = 5$ by $1/2$ yields $X - 3Y = 5/2$. Note that this operation has no influence on the abstraction of the constraint.

[3] The right-hand side is assumed to be a compound term to avoid an overlap with case (a).

[4] This is convenient when describing the derivation process at the abstract level afterwards.

Definition 5.3.6 (Derivation rules (unification constraints))

u1. *(elimination of X by confrontation)*

$$\left.\begin{array}{l} X = \mathcal{T}_1[Y] \text{ with } Y = \mathcal{T}_1[Y]/s_1 \\ X = \mathcal{T}_2[Z] \text{ with } Z = \mathcal{T}_2[Z]/s_1.s_2 \end{array}\right\} \Rightarrow \; Y = \mathcal{T}_3[Z] \text{ with } Z = \mathcal{T}_3[Z]/s_2$$

(note: the right-hand sides $\mathcal{T}_1[Y]$ and $\mathcal{T}_2[Z]$ must be unifiable.)

A variant of this rule may be used to derive that the variable Y is non-free, i.e.

$$\left.\begin{array}{l} X = \mathcal{T}_1[Y] \text{ with } Y = \mathcal{T}_1[Y]/s_1 \\ X = \mathcal{T}_2[Z] \text{ with } Z = \mathcal{T}_2[Z]/s_1.s_2 \;\mathscr{E}\; s_2 \neq \varepsilon \end{array}\right\} \Rightarrow \; Y = \mathcal{T}_3^+[Z] \text{ with } Z = \mathcal{T}_3^+[Z]/s_2$$

u2. *(elimination of X by substitution)*

$$\left.\begin{array}{l} X = \mathcal{T}_1[Z] \\ Y = \mathcal{T}_2^+[X] \end{array}\right\} \Rightarrow \; Y = \mathcal{T}_3^+[Z]$$

For the numerical part, variable elimination is again the fundamental operation; it is performed by computing a linear combination of two primitive equations. In this case however, the elimination of one variable may cause simultaneous elimination of other variables. In general, a subexpression is eliminated. For example, consider $X - Y - Z = 6 \wedge T - 2Y - 2Z = 3$; eliminating Y causes Z to be removed as well, yielding $X - T/2 = 9/2$. Note that the subexpressions $-Y - Z$ in the first equation and $-2Y - 2Z$ in the second equation are linearly dependent.

Definition 5.3.7 (Derivation rules (numerical constraints))

n1. *(elimination of e)*

$$\left.\begin{array}{l} m * e = d' \\ n * e = d'' \end{array}\right\} \Rightarrow \; d_1 = d_2$$

where

- *$vars(d_1) \cap vars(d_2) = \emptyset$, $vars(e) \cap vars(d') = \emptyset$, $vars(e) \cap vars(d'') = \emptyset$;*
- *d' and d'' can be split resp. into $d_1' + d_2'$ and $d_1'' + d_2''$, such that $d_1 \doteq (-a * n) * d_1' \dot{+} (a * m) * d_1''$ and $d_2 \doteq (a * n) * d_2' \dot{+} (-a * m) * d_2''$. So, $vars(d_1', d_1'') = vars(d_1)$ and $vars(d_2', d_2'') = vars(d_2)$. (note : since $vars(d_1) \cap vars(d_2) = \emptyset$, also $vars(d_1') \cap vars(d_2') = \emptyset$ and $vars(d_1'') \cap vars(d_2') = \emptyset$, $vars(d_1') \cap vars(d_2'') = \emptyset$ and $vars(d_1'') \cap vars(d_2'') = \emptyset$).*
- *m, n and a are non-zero numbers.*

The factor a is a normalisation factor to transform the derived equation to normal form (with its leftmost coefficient being 1). The derivation rule emphasizes that (1) a multiple of the same subexpression e occurs in each of the two starting constraints; (2) d_1 and d_2 can both be written as a linear combination of subexpressions of the starting constraints and (3) the subexpression d_1 in $d_1 = d_2$ is relevant to the following derivation step (i.e. it will be eliminated in that step). Note that this rule subsumes the steps in the computation of the solved form of a numerical constraint (i.e. multiplication of an equation by a non-zero number and addition of two equations; cf. Section 2.2).

The unification part and numerical part of a constraint in general interact. The interaction could be captured by the following rules :

m1.
$X = Y \Rightarrow X - Y = 0$ (only if X and Y are not of type Herbrand in C);
$X = n$ can directly be considered as a primitive numerical constraint.

m2.
$X - Y = 0 \Rightarrow X = Y$;
$X = n$ can directly be considered as a primitive unification constraint[5].

Rule m1 passes information from the unification part to the numerical part and corresponds to step 2 in Algorithm 2.2.2; rule m2 passes information in the other direction (step 4 in the algorithm). Adding these rules explicitly would only divert the attention from the essential derivation steps involving variable elimination. So, in the rest of this section, no distinction will be made between $X - Y = 0$ and $X = Y$ in case X and Y are not of type Herbrand[6]. The form in which the equation is written depends on how it is used : if it is used as starting constraint in rule $u1$ or $u2$, it is written as $X = Y$ and interpreted as a unification constraint; if it is used as starting constraint in rule $n1$, it is interpreted as a numerical constraint and may be written in both forms, emphasizing only one variable X or Y or the complete expression $X - Y$.

The derivation of each primitive constraint c in $sform(C)$, using the above derivation rules, can be represented by a finite *derivation tree.*

Definition 5.3.8 (Derivation tree)
Let $C \in SCons$ and $c \in sform(C)$. A derivation tree for c over C is defined as follows.

- *Every node in the tree is a primitive constraint $c' \in C^*$. A node c_0 has subnodes c_1 and c_2 if one of the derivation rules u1, u2 or n1 allows to obtain c_0 from c_1 and c_2. The derivation step for c_0 must not reintroduce variables that are eliminated in subtrees for deriving c_1 and c_2.*
- *The root of the tree is c.*
- *The leaves of the tree are primitive constraints of C.*

Example 5.3.3
Let $C \equiv X = f(Z) \wedge X = f(Y) \wedge W = g(X) \wedge T - U - 2Y + 2Z = 0 \wedge V + 3U = 5$.
Then

$$\begin{aligned}
sform(unif^*(C)) &= X = f(Z) \wedge Y = Z \wedge W = g(f(Z)) \wedge T = U \\
sform(num^*(C)) &= T - U = 0 \wedge Y - Z = 0 \wedge V + 3U = 5 \\
&\quad \textit{(solved form with parameters U and Z)} \\
\textit{or} &= Y - Z = 0 \wedge T + 1/3V = 5/3 \wedge U + 1/3V = 5/3 \\
&\quad \textit{(solved form with parameters Z and V)} \\
\textit{or} &= \ldots
\end{aligned}$$

The derivation trees in Figure 5.2 show how some primitive constraints in the solved form are obtained (constraints are written in normal form).

[5]Note that $aX_1 - aY_1 = 0$ or $aX = b$ with $a \neq 1$ cannot occur since each equation is assumed to be written in normal form.

[6]If X and Y are Herbrand variables in a constraint C, then $X = Y$ in C can only be considered as a unification constraint.

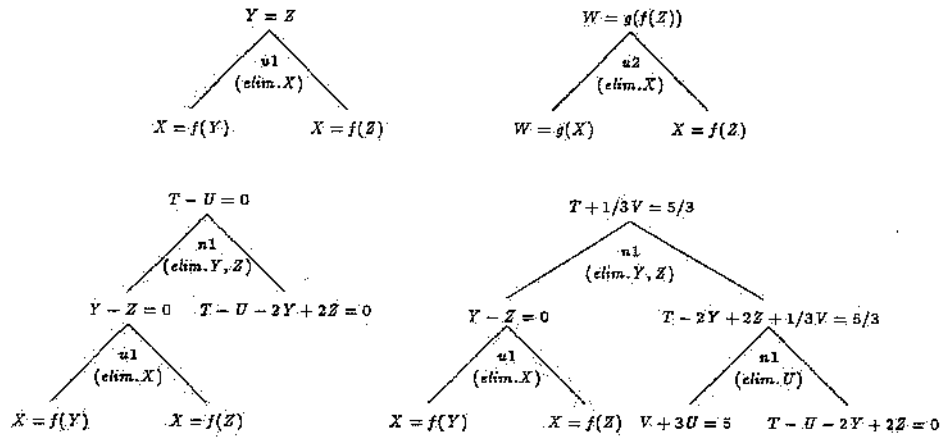


Figure 5.2: Derivation trees

So far, we have described the derivation process at the concrete level. At the abstract level, each basic dependency $A \in \alpha(C)$ (i.e. a variable set obtained before applying the closure operation in Definitions 5.2.10, 5.2.12 and 5.2.13) is obtained via one or more primitive constraints in $sform(C)$. We say that one constraint yields a *direct* dependency A and two or more constraints yield an *indirect* dependency A. In order to reflect the derivation of A, the concept of a derivation tree is extended to a *labelled derivation tree*. Each node in a labelled derivation tree

- contains a *constraint* $c \in C^*$ and
- is adorned with a *set label* $B \subseteq vars(c)$ such that B is a basic dependency in $\alpha(c)$.

A node is denoted by $(c; B)$. For a primitive unification constraint c of the form $X = t$, B belongs to $\{\{X\} \mid t \text{ is a non-variable}\} \cup \{\{X, Y\} \mid Y \in vars(t)\}$. For a primitive numerical constraint c, B is $vars(c)$. The constraints show how the derivation is performed at the concrete level, whereas the set labels mimic this derivation at the abstract level.
The derivation rules are adapted accordingly to labelled derivation rules. A direct dependency $A \in \alpha(C)$ will be at the root of a labelled derivation tree. Two extra derivation rules (obtained out of Definitions 5.2.12 and 5.2.13) are needed to infer an indirect dependency A from the root information of two or more labelled derivation trees.

Definition 5.3.9 (Labeled derivation rules (direct dependencies))

lu1. *(elimination of X)*

$$\left.\begin{array}{l}(X = T_1[Y] \text{ with } Y = T_1[Y]/s_1; \{X, Y\}) \\ (X = T_2[Z] \text{ with } X = T_2[Z]/s_1.s_2; \{X, Z\})\end{array}\right\} \Rightarrow$$
$$(Y = T_3[Z] \text{ with } Z = T_3[Z]/s_2; \{Y, Z\})$$

A variant of this rule may be used to derive that the variable Y is non-free, i.e.

$$\left.\begin{array}{l}(X = T_1[Y] \text{ with } Y = T_1[Y]/s_1; \{X, Y\}) \\ (X = T_2[Z] \text{ with } Z = T_2[Z]/s_1.s_2 \text{ and } s_2 \neq \varepsilon; \{X\})\end{array}\right\} \Rightarrow$$
$$(Y = T_3^+[Z] \text{ with } Z = T_3^+[Z]/s_2; \{Y\})$$

lu2. *(elimination of X)*

$$\left.\begin{array}{l}(X = \mathcal{T}_1[Z]; \{X, Z\}) \\ (Y = \mathcal{T}_2^+[X]; \{Y, X\})\end{array}\right\} \Rightarrow \; (Y = \mathcal{T}_3^+[Z]; \{Y, Z\})$$

ln1. *(elimination of e)*

$$\left.\begin{array}{l}(m * e = d'; vars(e, d')) \\ (n * e = d''; vars(e, d''))\end{array}\right\} \Rightarrow \; (d_1 = d_2; vars(d_1, d_2))$$

where

- $vars(d_1) \cap vars(d_2) = \emptyset$, $vars(e) \cap vars(d') = \emptyset$, $vars(e) \cap vars(d'') = \emptyset$;
- *d' and d'' can be split resp. into $d_1' + d_2'$ and $d_1'' + d_2''$, such that* $d_1 \doteq (-a * n) * d_1' \dotplus (a * m) * d_1''$ *and* $d_2 \doteq (a * n) * d_2' \dotplus (-a * m) * d_2''$. *So,* $vars(d_1', d_1'') = vars(d_1)$, $vars(d_2', d_2'') = vars(d_2)$, $vars(d', d'') = vars(d_1, d_2)$. *(note : since* $vars(d_1) \cap vars(d_2) = \emptyset$, *also* $vars(d_1') \cap vars(d_2') = \emptyset$ *and* $vars(d_1'') \cap vars(d_2') = \emptyset$, $vars(d_1') \cap vars(d_2'') = \emptyset$ *and* $vars(d_1'') \cap vars(d_2'') = \emptyset$ *).*
- *m, n and a are non-zero numbers.*

Definition 5.3.10 (Labeled derivation rules (indirect dependencies))

li1. *(elimination of Z)*

$$\left.\begin{array}{l}(X = \mathcal{T}_1[Z]; \{X, Z\}) \\ (Y = \mathcal{T}_2[Z]; \{Y, Z\})\end{array}\right\} \Rightarrow \; \{X, Y\}$$

li2. *(elimination of $Y_1, \ldots, Y_m$)*

$$\left.\begin{array}{l}(X_1 = \mathcal{T}_1^+[Y_1]; \{X_1, Y_1\}) \\ \quad \ldots \\ (X_m = \mathcal{T}_m^+[Y_m]; \{X_m, Y_m\}) \\ (a_1 Y_1 + \ldots + a_m Y_m = d; \{Y_1, \ldots, Y_m\} \cup vars(d))^7\end{array}\right\} \Rightarrow \; \{X_1, \ldots, X_m\} \cup vars(d)$$

The Y_i are distinct variables; the X_i are not necessarily distinct.

Definition 5.3.11 (Labeled derivation tree)
Let $C \in SCons$. Let $c \in sform(C)$ with A a basic dependency in $\alpha(c)$. A labelled derivation tree $\mathbb{T}$ for $(c; A)$ over C is defined as follows.

- *Every node in $\mathbb{T}$ is $(c'; B)$ where $c' \in C^*$ and B is a basic dependency in $\alpha(c')$. A node $(c_0; B_0)$ has subnodes $(c_1; B_1)$ and $(c_2; B_2)$ if one of the labelled derivation rules lu1, lu2 or ln1 allows to obtain $(c_0; B_0)$ from $(c_1; B_1)$ and $(c_2; B_2)$.*
- *The root of $\mathbb{T}$ is $(c; A)$.*
- *A leaf of $\mathbb{T}$ is labelled with $(c'; B)$ where $c' \in C$.*

Eliminated variables must not be reintroduced.

Definition 5.3.12 (nodes($\mathbb{T}$) – leaves($\mathbb{T}$))
Let $\mathbb{T}$ be a labelled derivation tree. Then nodes($\mathbb{T}$) and leaves($\mathbb{T}$) denote respectively the set of all nodes and all leaf nodes in $\mathbb{T}$.

[7] $a_1 Y_1 + \ldots + a_m Y_m = d$ is not of the form $Y_1 - Y_2 = 0$, cf. Definition 5.2.13.

Definition 5.3.13 ($\mathbb{T}_{(c;B)}$)
Let $\mathbb{T}$ be a labelled derivation tree with $(c;B) \in nodes(\mathbb{T})$. Then $\mathbb{T}_{(c;B)}$ denotes the subtree of $\mathbb{T}$ rooted at $(c;B)$.

Example 5.3.4
Let $C \equiv X = f(Y,Z) \wedge Y = g(T) \wedge Y = W$. Then $sform(C) \equiv X = f(g(T),Z) \wedge Y = g(T) \wedge W = g(T)$. Examples of basic dependencies in $\alpha(C)$ that are established by the primitive constraints in $sform(C)$ are $\{W\}$ and $\{X,T\}$; the labelled derivation trees capturing the derivation process are shown in Figure 5.3.

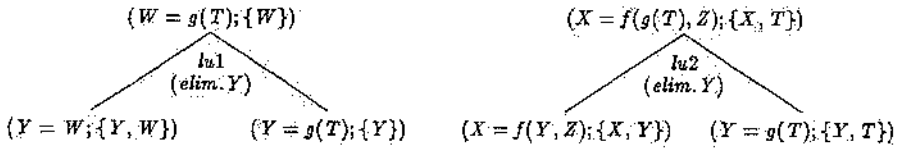


Figure 5.3: Labeled derivation trees

Let $C \equiv X = f(Z) \wedge X = f(Y) \wedge T - U - 2Y + 2Z = 0$. Then $sform(unif^*(C)) \equiv X = f(Z) \wedge Y = Z \wedge T = U$ and $sform(num^*(C)) \equiv T - U = 0 \wedge Y - Z = 0$. The labelled derivation tree for $(T - U = 0;\{T,U\})$ is shown in Figure 5.4.

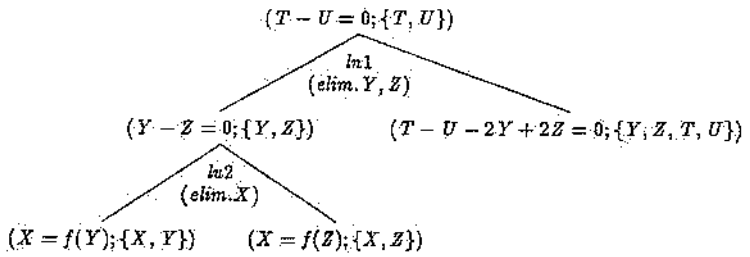


Figure 5.4: Labeled derivation tree for $(T - U = 0;\{T,U\})$

Finally, we consider the derivation of an indirect dependency. Let $C \equiv X = f(A,B) \wedge A + B - T = 3$; C establishes the indirect dependency $\{X,T\}$. The derivation is illustrated in Figure 5.5.

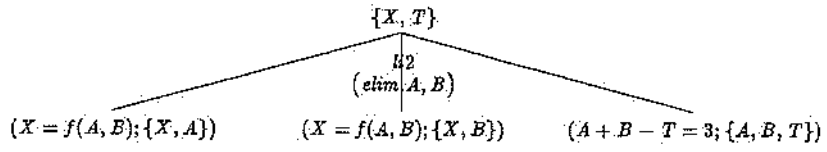


Figure 5.5: Labeled derivation tree for $\{X,T\}$

In the rest of this section, attention is focussed on derivations from a satisfiable concrete constraint of the form $C_1 \wedge C_2$ with $C_1, C_2 \in SCons$. Let $\mathbb{T}$ be a labelled derivation tree over

$C_1 \wedge C_2$. Each constraint in a leaf of $\mathbb{T}$ belongs to either C_1 or C_2. Hence, the set label of a leaf belongs to $\alpha(C_1)$ or $\alpha(C_2)$. For a non-leaf node $(c; B)$ in $\mathbb{T}$, the constraint c belongs to $(C_1 \wedge C_2)^*$ and its set label B to $\alpha(C_1 \wedge C_2)$. The essence in proving the safety of abstract conjunction consists of showing that B also belongs to $\alpha(C_1) \wedge \alpha(C_2)$. In other words, the result of the abstract conjunction of $\alpha(C_1)$ and $\alpha(C_2)$ (i.e. $\alpha(C_1) \wedge \alpha(C_2)$) should cover the abstraction of the concrete conjunction $C_1 \wedge C_2$. This amounts to showing that B can be constructed from a dependency in $\alpha(C_1)$ and/or a dependency in $\alpha(C_2)$. More precisely, B can be written as $T^1 \cup T^2$ with $T^1 \cup T \in \alpha(C_1)$ and $T^2 \cup T \in \alpha(C_2)$[8] (note: T^1 and T^2 are not necessarily disjoint). Both $T^1 \cup T$ and $T^2 \cup T$ are derived from $leaves(\mathbb{T}_{(c;B)})$. This leaves-set can be divided in two disjoint subsets : $lvs^1 = \{(c_1^1; S_1^1), \ldots, (c_m^1; S_m^1)\}$, related to C_1 (i.e. $c_i^1 \in C_1$, $S_i^1 \in \alpha(C_1)$), and $lvs^2 = \{(c_1^2; S_1^2), \ldots, (c_n^2; S_n^2)\}$, related to C_2. Then $T^1 \cup T \subseteq S_1^1 \cup \ldots \cup S_m^1$ and $T^2 \cup T \subseteq S_1^2 \cup \ldots \cup S_n^2$. In general, the derivation of $(c; B)$ involves leaves of both C_1 and C_2, i.e. $T^1 \cup T$ (or lvs^1) and $T^2 \cup T$ (or lvs^2) are not empty and $B \in \alpha(C_1) \oplus \alpha(C_2)$. If $T^2 \cup T$ (or lvs^2) is empty, then $B \in \alpha(C_1)$; if $T^1 \cup T$ (or lvs^1) is empty, then $B \in \alpha(C_2)$. The set T is a subset of the variables that are eliminated while deriving $(c; B)$.

Example 5.3.5
Let $C_1 \equiv U = f(X)$ *and* $C_2 \equiv U = f(Z) \wedge Z - Y - 2X = 3$; $\alpha(C_1) = \{\{U\}, \{U, X\}\}$ *and* $\alpha(C_2) = \{\{U\}, \{U, Z\}, \{Z, Y, X\}, \{U, Y, X\}, \{U, Z, Y, X\}\}$.

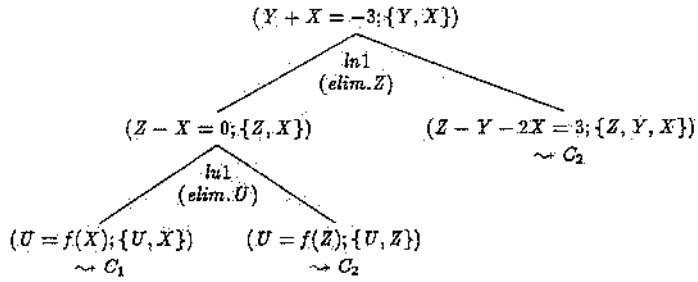


Figure 5.6: Labeled derivation tree for $(Y + X = -3; \{Y, X\})$

Then $sform(unif^*(C_1 \wedge C_2)) \equiv U = f(X) \wedge Z = X$ *and* $sform(num^*(C_1 \wedge C_2)) \equiv Z - X = 0 \wedge Y + X = -3$. *The labelled derivation tree for* $(Y + X = -3; \{Y, X\})$ *is given in Figure 5.6.*
$lvs^1 = \{(U = f(X); \{U, X\})\}$ *and*
$lvs^2 = \{(U = f(Z); \{U, Z\}), (Z - Y - 2X = 3; \{Z, Y, X\})\}$.
$\{Y, X\}$ *can be written as* $T^1 \cup T^2 = \{X\} \cup \{Y, X\}$ *with* $T^1 \cup T = \{X\} \cup \{U\} \in \alpha(C_1)$ *and* $T^2 \cup T = \{Y, X\} \cup \{U\} \in \alpha(C_2)$. *Note that* $T^1 \cup T \subseteq \{U, X\}$ *($\{U, X\}$ is the union of the set labels of the leaves related to C_1) and* $T^2 \cup T \subseteq \{U, Z, Y, X\}$ *($\{U, Z, Y, X\}$ is the union of the set labels of the leaves related to C_2). Also* $T = \{U\}$ *is a subset of the variables that are eliminated while deriving* $(Y + X = -3; \{Y, X\})$.

Let $(c; B)$ be a node in a labelled derivation tree $\mathbb{T}$ over $C_1 \wedge C_2$. For each non-numerical variable X or numerical subterm e in c (a special case is $e \equiv X$ where X is numerical in

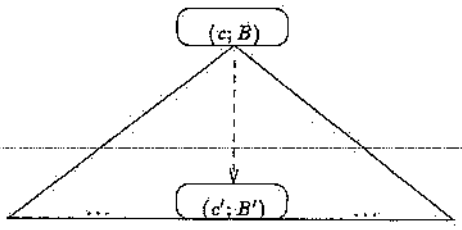
[8] In the remainder of this section, the superscripts 1 and 2 denote resp. an association with C_1 and C_2.

$C_1 \wedge C_2$), the Lemmas 5.3.1 and 5.3.2 allow to infer information respectively on the leaf of $\mathbb{T}_{(c;B)}$ that contains X or on the subset of leaves of $\mathbb{T}_{(c;B)}$ that contribute to c. The lemmas are illustrated in Figures 5.7 and 5.8.

Lemma 5.3.1
Let $C_1 \wedge C_2 \in SCons$. Let $\mathbb{T}$ be a labelled derivation tree over $C_1 \wedge C_2$ with $(c; B) \in nodes(\mathbb{T})$ such that $X \in B$ ($\subseteq vars(c)$) and X is non-numerical in $C_1 \wedge C_2$.
Then there exists a unique $(c'; \ldots) \in leaves(\mathbb{T}_{(c;B)})$ such that

1. *if c is of the form $X = \mathcal{T}_1[W]$, then c' is either of the form $X = \mathcal{T}_2[V]$, with W and V having compatible selectors, or of the form $V = \mathcal{T}_3[X]$;*
2. *if c is of the form $W = \mathcal{T}_1[X]$, then c' is of the form $V = \mathcal{T}_2[X]$.*

Let $B = T^1 \cup T^2$ with $T^1 \cup T \in \alpha(C_1)$ and $T^2 \cup T \in \alpha(C_2)$. If the considered X is in T^k (k either 1 or 2), then it follows that $c' \in C_k$. The link between X and the other variables in T^k is established via V in c'. This is implied by the construction of $\mathbb{T}$.



$X \in B$; c is of the form $X = \mathcal{T}_1[W]$ or $W = \mathcal{T}_2[X]$
c' is of the form $X = \mathcal{T}_3[V]$ or $V = \mathcal{T}_4[X]$

Figure 5.7: Illustration of Lemma 5.3.1 : c contains the non-numerical variable X.

PROOF
If $c \in leaves(\mathbb{T})$, then the result is trivial : $c' \equiv c$. Otherwise, the proof is by induction on the depth of the tree $\mathbb{T}_{(c;B)}$. Notice that all primitive constraints used in the derivation of c that contain X will be unification constraints, since X is non-numerical; so the labelled derivation rules that may be involved are *lu*1 and *lu*2.

Base case : $\mathbb{T}_{(c;B)}$ is of depth 1 (i.e. c is obtained via a single derivation step).

1. c is of the form $X = \mathcal{T}_1[W]$

   $X = \mathcal{T}_1[W]$ can be obtained via derivation rule *lu*1 or *lu*2. Rule *lu*1 implies that $\exists c' \equiv (V = \mathcal{T}_3[X]) \in leaves(\mathbb{T}_{(c;B)})$ where c' is one of the starting constraints in the rule. Note that c' is a leaf since $\mathbb{T}_{(c;B)}$ is assumed to be of depth 1. Rule *lu*2 implies that $\exists c' \equiv (X = \mathcal{T}_2[V]) \in leaves(\mathbb{T}_{(c;B)})$. The other starting constraint in *lu*2 is of the form $V = \mathcal{T}_4[W]$, so W and V have compatible selectors within the right-hand side terms of c and c'.

2. c is of the form $W = \mathcal{T}_1[X]$

   $W = \mathcal{T}_1[X]$ can be obtained via derivation rule *lu*1 or *lu*2. These rules imply that $\exists c' \equiv (V = \mathcal{T}_2[X]) \in leaves(\mathbb{T}_{(c;B)})$.

Induction : $\mathbb{T}_{(c;B)}$ is of depth k $(k > 1)$.

1. c is of the form $X = \mathcal{T}_1[W]$

   c can be obtained via derivation rule *lu*1 or *lu*2. Rule *lu*1 implies that $\exists c'' \equiv (V' = \mathcal{T}_4[X])$. Rule *lu*2 implies that $\exists c'' \equiv (X = \mathcal{T}_5[V'])$. W and V' have compatible selectors within the right-hand side terms of c and c''. (1)
   Since c'' is a subnode of c, c'' is the root of a derivation tree of depth $\leq k-1$. Using the induction hypothesis, either $\exists c' \equiv (X = \mathcal{T}_2[V]) \in \mathit{leaves}(\mathbb{T}_{(c';B)})$ or $\exists c' \equiv (V = \mathcal{T}_3[X]) \in \mathit{leaves}(\mathbb{T}_{(c';B)})$ and $\mathit{leaves}(\mathbb{T}_{(c';B)}) \subset \mathit{leaves}(\mathbb{T}_{(c;B)})$. In the former case, V and V' have compatible selectors within the right-hand side terms of c'' and c'; together with (1), this implies that W and V have compatible selectors within the right-hand side terms of c and c'.

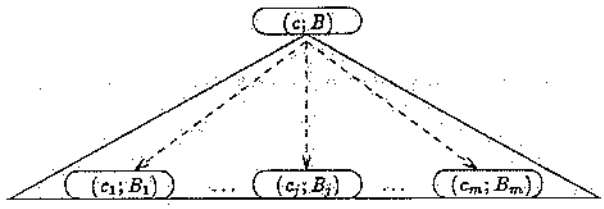
2. c is of the form $W = \mathcal{T}_1[X]$

   c can be obtained via derivation rule *lu*1 or *lu*2. These rules imply that $\exists c'' \equiv (V' = \mathcal{T}_3[X])$. Since c'' is a subnode of c, c'' is the root of a derivation tree of depth $\leq k-1$. Using the induction hypothesis, $\exists c' \equiv (V = \mathcal{T}_2[X]) \in \mathit{leaves}(\mathbb{T}_{(c';B)}) \subset \mathit{leaves}(\mathbb{T}_{(c;B)})$.

□

Lemma 5.3.2
Let $C_1 \wedge C_2 \in SCons$. Let $\mathbb{T}$ be a labelled derivation tree over $C_1 \wedge C_2$ with $(c;B) \in nodes(\mathbb{T})$ such that c contains the numerical term e and $vars(e) \subseteq B$.
Then there exists $\{(c_1;\ldots),\ldots,(c_m;\ldots)\} \subseteq \mathit{leaves}(\mathbb{T}_{(c;B)})$ such that each c_j is of the form $(e_j = e_j'$ or $V_j = \mathcal{T}_j[e_j]$ (in the latter case, $e_j \equiv X_j \in vars(e))$ and $e \doteq b_1 e_1 + \ldots + b_m e_m$ with $vars(e_j) \subseteq vars(e)$; $(c_1;\ldots),\ldots,(c_m;\ldots)$ are the only leaves of $\mathbb{T}_{(c;B)}$ that contain variables of e.

Let $B = T^1 \cup T^2$ with $T^1 \cup T \in \alpha(C_1)$ and $T^2 \cup T \in \alpha(C_2)$. If $X \in vars(e)$ occurs only within T^1 (resp. T^2), then each c_j containing X belongs to C_1 (resp. C_2); if $X \in T^1 \cap T^2$, then there is at least one $c_i \in C_1$ and at least one $c_j \in C_2$ containing X. The link or dependency between e and the other variables in T^1 or T^2 is established via the e_j' and V_j in the c_j. This is implied by the construction of $\mathbb{T}$.



$vars(e) \subseteq B$; c contains e
c_j contains e_j (c_j is of the form $e_j = e_j'$ or $V_j = \mathcal{T}_j[e_j]$)

Figure 5.8: Illustration of Lemma 5.3.2 : c contains e and c_j contains e_j.

PROOF
If $c \in leaves(\mathbb{T})$, then the result is trivial : $c_1 \equiv c$. Otherwise, the proof is by induction on the depth of the tree $\mathbb{T}_{(c;B)}$.

Base case : $\mathbb{T}_{(c;B)}$ is of depth 1 (i.e. c is obtained via a single derivation step).

If c is of the form $W = \mathcal{T}_0[e]$ (with $e \equiv X$), then it can be obtained via rule *lu*1 or *lu*2. Each of these rules implies that there is a starting constraint c_1 of the form $V = \mathcal{T}_1[e]$. Since $\mathbb{T}_{(c;B)}$ is assumed to be of depth 1, $(c_1;\ldots) \in leaves(\mathbb{T}_{(c;B)})$. Alternatively, c is of the form $e = d$ and is derived via rule *ln*1. It implies that $\exists (c_1;\ldots),(c_2;\ldots) \in leaves(\mathbb{T}_{(c;B)})$ where c_1 contains e_1 and c_2 contains e_2 such that $e \doteq (-a*n)*e_1 \dot{+} (a*m)*e_2$ with $vars(e) = vars(e_1,e_2)$. Putting the emphasis on e_1 and e_2, c_1 and c_2 can be rewritten as $e_1 = e_1'$ and $e_2 = e_2'$.
Whichever rule is applied, the result can be summarised as: $\exists\{(c_1;\ldots),\ldots,(c_m;\ldots)\} \subseteq leaves(\mathbb{T}_{(c;B)})$ such that c_j is of the form $(e_j = e_j'$ or $V_j = \mathcal{T}_j[e_j])$ and $e \doteq b_1 e_1 \dot{+} \ldots \dot{+} b_m e_m$ with $vars(e_j) \subseteq vars(e)$ (j in $1..m$).

Induction : $\mathbb{T}_{(c;B)}$ of depth k ($k > 1$)

Rules *lu*1 and *lu*2 allow to obtain a constraint c of the form $W = \mathcal{T}_0[e]$ with $e \equiv X$. Then there is a starting constraint c_1' of the form $V = \mathcal{T}_1[e]$; c_1' is a subnode of c, so the depth of $\mathbb{T}_{(c_1';B)}$ is $\leq k-1$. Using the induction hypothesis, $\exists\{(c_1;\ldots),\ldots,(c_m;\ldots)\} \subseteq leaves(\mathbb{T}_{(c_1';B)}) \subset leaves(\mathbb{T}_{(c;B)})$ such that each c_j is of the form $(e_j = e_j'$ or $V_j = \mathcal{T}_j[e_j])$ and $e \doteq b_1 e_1 \dot{+} \ldots \dot{+} b_m e_m$ and $vars(e_j) = X$.
Alternatively, rule *ln*1 allows to obtain a constraint c of the form $e = d$ ($X = d$ is a special case). It implies that $\exists (c_1';\ldots),(c_2';\ldots) \in leaves(\mathbb{T}_{(c;B)})$ where c_1' contains e_1 and c_2' contains e_2 such that $e \doteq (-a*n)*e_1 \dot{+} (a*m)*e_2$ with $vars(e) = vars(e_1,e_2)$. Since c_1' and c_2' are subnodes of c, the depth of $\mathbb{T}_{(c_1';\ldots)}$ and $\mathbb{T}_{(c_2';\ldots)}$ is $\leq k-1$. Using the induction hypothesis,

- $\exists S_1 = \{(c_{11};\ldots),\ldots,(c_{1p};\ldots)\} \subseteq leaves(\mathbb{T}_{(c_1';B)}) \subset leaves(\mathbb{T}_{(c;B)})$ such that each c_{1i} is of the form $(e_{1i} = e_{1i}'$ or $V_{1i} = \mathcal{T}_{1i}[e_{1i}])$ and $e_1 \doteq b_{11}e_{11} \dot{+} \ldots \dot{+} b_{1p}e_{1p}$ with $vars(e_{1i}) \subseteq vars(e_1) \subseteq vars(e)$;
- $\exists S_2 = \{(c_{21};\ldots),\ldots,(c_{2q};\ldots)\} \subseteq leaves(\mathbb{T}_{(c_2';B)}) \subset leaves(\mathbb{T}_{(c;B)})$ such that each c_{2j} is of the form $(e_{2j} = e_{2j}'$ or $V_{2j} = \mathcal{T}_{2j}[e_{2j}])$ and $e_2 \doteq b_{21}e_{21} \dot{+} \ldots \dot{+} b_{2q}e_{2q}$ with $vars(e_{2j}) \subseteq vars(e_2) \subseteq vars(e)$.

So, e can be written as $e \doteq (-a*n*b_{11})e_{11} \dot{+} \ldots \dot{+} (-a*n*b_{1p})e_{1p} \dot{+} (a*m*b_{21})e_{21} \dot{+} \ldots \dot{+} (a*m*b_{2q})e_{2q}$ and $S_1 \cup S_2$ is the set of leaves of $\mathbb{T}_{(c;B)}$ that contribute to e in c.
Whichever rule is applied, the result can be summarised as : $\exists\{(c_1;\ldots),\ldots,(c_m;\ldots)\} \subseteq leaves(\mathbb{T}_{(c;B)})$ such that each c_j is of the form $(e_j = e_j'$ or $V_j = \mathcal{T}_j[e_j])$ (in the latter case $e_j \equiv Y_j \in vars(e)$) and $e \doteq b_1 e_1 \dot{+} \ldots \dot{+} b_m e_m$ with $vars(e_j) \subseteq vars(e)$. □

Let V be the set of *all* numerical variables that are involved in the derivation of $(c;B)$ and let V_{sub} be some subset of V. The above lemma can be extended to consider how the derivation of $(c;B)$ manipulates the variables of V_{sub}, starting from their occurrences in $leaves(\mathbb{T}_{(c;B)})$. The extension consists of considering not only (some of) the numerical variables occurring in B (being $vars(e)$ in Lemma 5.3.2) but also (some of) the numerical variables that are eliminated during the derivation of $(c;B)$.

Lemma 5.3.2 extended

Let V be the set of all *numerical variables that are involved in the derivation of $(c;B)$. Let $V_{sub} \subseteq V$ be the set of variables that is considered and let e be the subexpression in c that contains all V_{sub} variables occurring in B ($V_{sub} \cap B = vars(e)$).*
Then e can be written as $e \doteq \sum_i b_i e_i$ where $vars(e) \subseteq \bigcup_i vars(e_i)$, $\bigcup_i vars(e_i) = V_{sub}$ and the e_i are all the subexpressions in leaves$(\mathbb{T}_{(c;B)})$ that contain variables of V_{sub}. For a $X \in vars(e)$, the sum of its coefficients in $\sum_i b_i e_i$ (which equals its coefficient in e) is not zero; for each $X \in V_{sub} \setminus vars(e)$, the sum of its coefficients in $\sum_i b_i e_i$ is zero.

PROOF
Similar to proof of Lemma 5.3.2. □

Let $\mathbb{T}$ be a labelled derivation tree over $C_1 \wedge C_2$ that contains a node $(c_0;B_0)$ with subnodes $(c_1;B_1)$ and $(c_2;B_2)$. Assume that $B_1 = T_1^1 \cup T_1^2$ with[9] $T_1^1 \cup T \in \alpha(C_1)$ and $T_1^2 \cup T \in \alpha(C_2)$; in the sequel, $T_1^k \cup T$ is called the *C_k-part* of B_1 (k in 1..2). Similarly, $B_2 = T_2^1 \cup T_2^2$ with $T_2^1 \cup T' \in \alpha(C_1)$ and $T_2^2 \cup T' \in \alpha(C_2)$. This implies that B_1 and B_2 belong to $\alpha(C_1) \wedge \alpha(C_2)$. The aim is to show that B_0 can be written as $B_0 = S^1 \cup S^2$ with $S^1 \cup S \in \alpha(C_1)$ and $S^2 \cup S \in \alpha(C_2)$, so also $B_0 \in \alpha(C_1) \wedge \alpha(C_2)$. The idea is to construct the S-sets from the T-sets. The leaves in $\mathbb{T}_{(c_1;B_1)}$ and $\mathbb{T}_{(c_2;B_2)}$ that are related to C_1 and that are used to establish the C_1-parts of B_1 and B_2 can be combined to establish a new dependency $S^1 \cup S \subseteq T_1^1 \cup T \cup T_2^1 \cup T'$ in $\alpha(C_1)$. Similarly, the leaves in $\mathbb{T}_{(c_1;B_1)}$ and $\mathbb{T}_{(c_2;B_2)}$ that are related to C_2 and that are used to establish the C_2-parts of B_1 and B_2 can be combined to a new link[10] $S^2 \cup S \subseteq T_1^2 \cup T \cup T_2^2 \cup T'$ in $\alpha(C_2)$. In case of "$\subset$", the C_k-parts (k in 1..2) of B_1 and B_2 are short-circuited into a smaller dependency : this involves the elimination of a variable or numerical term that is shared by c_1 and c_2 and that appears only either in the C_1-parts or in the C_2-parts. This elimination process is captured in Lemmas 5.3.3 and 5.3.4. It is based on the information on the leaves of $\mathbb{T}_{(c;B)}$ that contribute to the C_k-parts; this information is extracted via Lemmas 5.3.1 and 5.3.2. The above reasoning can easily be adapted if $(c_0;B_0)$ is replaced by an indirect dependency D that is obtained from a set of nodes $(c_i;B_i)$ with $1 \leq i \leq m$ (cf. Lemmas 5.3.3 and 5.3.5).

Example 5.3.6
Let $C_1 \equiv X = f(Z) \wedge V + 3U = 5$ and $C_2 \equiv X = f(Y) \wedge T - U - 2Y + 2Z = 0$; $\alpha(C_1) = close(\{\{X\},\{X,Z\},\{V,U\}\}$; $\alpha(C_2) = close(\{\{X\},\{X,Y\},\{T,U,Y,Z\},\{X,T,U,Z\}\}$. Then $sform(C_1 \wedge C_2)$ contains $T + 1/3\,V = 5/3$ (cf. Example 5.3.3); a labelled derivation tree for $(T + 1/3\,V = 5/3;\{T,V\})$ is shown in Figure 5.9.
B_1 can be written as $T_1^1 \cup T_1^2 = \{Z\} \cup \{Y\}$ with
$T_1^1 \cup T = \{Z\} \cup \{X\} \in \alpha(C_1)$ and
$T_1^2 \cup T = \{Y\} \cup \{X\} \in \alpha(C_2)$.
B_2 can be written as $T_2^1 \cup T_2^2 = \{V\} \cup \{T,Y,Z\}$ with
$T_2^1 \cup T' = \{V\} \cup \{U\} \in \alpha(C_1)$ and
$T_2^2 \cup T' = \{T,Y,Z\} \cup \{U\} \in \alpha(C_2)$.

[9] A superscript 1 or 2 denotes an association with resp. C_1 or C_2; a subscript 1 or 2 denotes an association with resp. the subtree $\mathbb{T}_{(c_1;B_1)}$ or $\mathbb{T}_{(c_2;B_2)}$.
[10] In the sequel, the term *link* is used as a synonym for *dependency*.

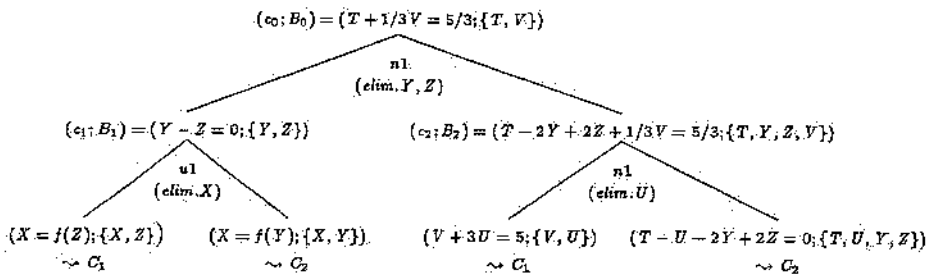


Figure 5.9: Labeled derivation tree for $(T + 1/3V = 5/3; \{T, V\})$

Then B_0 can be written as $S^1 \cup S^2 = \{V\} \cup \{T\}$ with
$S^1 \cup S = \{V\} \cup \{X, Z, U\} \in \alpha(C_1)$ *and*
$S^2 \cup S = \{T\} \cup \{X, Z, U\} \in \alpha(C_2)$.
Note that to obtain $S^2 \cup S$ the dependencies $\{X, Y\}$ and $\{T, U, Y, Z\}$ in $\alpha(C_2)$ are short-circuited to $\{X, T, U, Z\} \in \alpha(C_2)$ via elimination of Y.

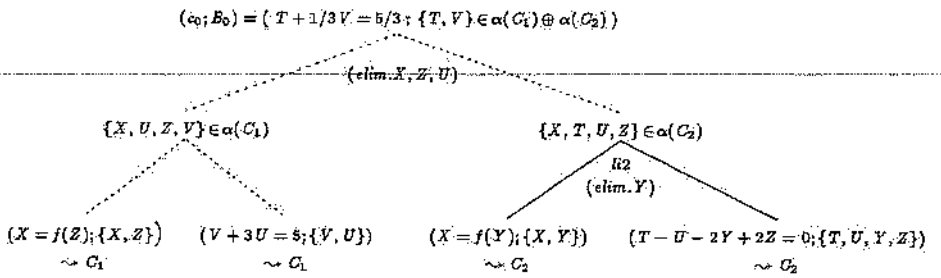


Figure 5.10: Reorganised tree

Intuitively, the derivation of the S-links from the T-links can be seen as reorganising the tree $\mathbb{T}_{(c_0;B_0)}$. Originally, the leaves related to C_1 and C_2 are spread over both subtrees of $(c_0; B_0)$. The reorganisation consists of attaching two new subtrees to $(c_0; B_0)$, such that one subtree contains all C_1-leaves (yielding the C_1-part of B_0) and the other subtree contains all C_2-leaves (yielding the C_2-part of B_0)[11]*. This is illustrated in Figure 5.10.*

Lemma 5.3.3 (Elimination of a non-numerical variable)
Let $C_1 \wedge C_2 \in SCons$. Assume that

1. *$(c_1; B_1)$ and $(c_2; B_2)$ are nodes in a labelled derivation tree $\mathbb{T}$ over $C_1 \wedge C_2$;*
2. *X is a non-numerical variable in $C_1 \wedge C_2$ and $X \in B_1 \cap B_2$;*
3. *$(c_1; B_1)$ and $(c_2; B_2)$ are used as starting nodes in derivation rule lu1 or lu2 to derive a node $(c_0; B_0)$ or in rule li1 to derive an indirect dependency B_0*[12]*; X is eliminated via this derivation step.*

[11] Note that the new tree is not really a labelled derivation tree, since the link between some subnodes and their father node cannot be described with one of the given derivation rules (since no variable elimination is involved). These links are represented by dashed lines in the figure.

[12] Rules *ln*1 and *li*2 are not applicable since X is non-numerical.

4. $B_1 = T_1^1 \cup T_1^2$ *with* $T_1^1 \cup T \in \alpha(C_1)$ *and* $T_1^2 \cup T \in \alpha(C_2)$;
   $B_2 = T_2^1 \cup T_2^2$ *with* $T_2^1 \cup T' \in \alpha(C_1)$ *and* $T_2^2 \cup T' \in \alpha(C_2)$;
   X *occurs only in* T_1^1 *and* T_2^1 *(i.e. only in the* C_1*-parts)*[13];

Then $(T_1^1 \cup T \cup T_2^1 \cup T') \setminus \{X\} \in \alpha(C_1)$, *i.e. the* C_1*-dependencies* $T_1^1 \cup T$ *and* $T_2^1 \cup T'$ *can be combined to a new* C_1*-dependency via elimination of* X.

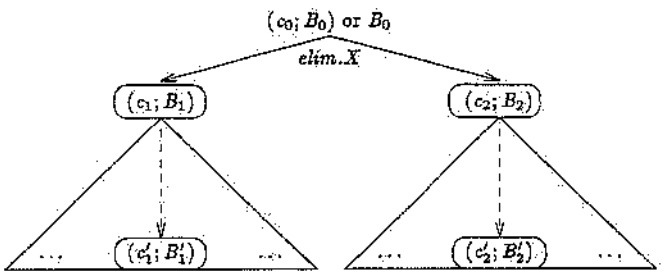


Figure 5.11: Situation given in Lemma 5.3.3 : c_1, c_2, c_1' and c_2' contain X.

PROOF
$(c_1; B_1)$ and $(c_2; B_2)$ are used as starting nodes in rule *lu*1, *lu*2 or *li*1; this rule determines the possible form of c_1 and c_2 :

- for *lu*1, c_1 is of the form $X = \mathcal{T}_1[W_1]$ and c_2 is of the form $X = \mathcal{T}_2[W_2]$, where W_1 and W_2 have compatible selectors within the right-hand side terms.
- for *lu*2, c_1 is of the form $W_1 = \mathcal{T}_1[X]$ and c_2 is of the form $X = \mathcal{T}_2[W_2]$.
- for *li*1, c_1 is of the form $W_1 = \mathcal{T}_1[X]$ and c_2 is of the form $W_2 = \mathcal{T}_2[X]$.

By Lemma 5.3.1, there exists a $(c_1'; \ldots) \in \mathit{leaves}(\mathbb{T}_{(c_1;B_1)})$ and a $(c_2'; \ldots) \in \mathit{leaves}(\mathbb{T}_{(c_2;B_2)})$ containing X; the form of c_l' is implied by the form of c_l (l in 1..2) :

- if c_l is of the form $X = \mathcal{T}_1[W_l]$ then c_l' is of the form $X = \mathcal{T}_2[V_l]$ where W_l and V_l have compatible selectors within the right-hand side terms, or c_l' is of the form $V_l = \mathcal{T}_3[X]$;
- if c_l is of the form $W_l = \mathcal{T}_1[X]$ then c_l' is of the form $V_l = \mathcal{T}_3[X]$.

The C_1-link between X and the other variables in $T_1^1 \cup T$ (resp. in $T_2^1 \cup T'$) is established via V_1 (resp. V_2); c_1' and c_2' belong to C_1 as X occurs only in the C_1-parts. Note that, in case of *lu*1, if $c_1' \equiv (X = \mathcal{T}_1[V_1])$ and $c_2' \equiv (X = \mathcal{T}_2[V_2])$ then V_1 and V_2 have compatible selectors since the pairs W_1 and V_1, W_2 and V_2, and W_1 and W_2 have compatible selectors. It has to be shown that the combination of c_1' and c_2' allows to short-circuit $T_1^1 \cup T$ and $T_2^1 \cup T'$ to $(T_1^1 \cup T \cup T_2^1 \cup T') \setminus \{X\}$ via elimination of X. Four cases can be distinguished:

1. $c_1' \equiv (X = \mathcal{T}_1[V_1])$, $c_2' \equiv (X = \mathcal{T}_2[V_2])$ where V_1 and V_2 have compatible selectors
   Rule *lu*1 allows to obtain an immediate dependency between V_1 and V_2 via elimination of X. Since the dependency between X and the other variables in $T_1^1 \cup T \in \alpha(C_1)$ and

[13] A similar reasoning holds if X occurs only in the C_2-parts.

in $T_2^1 \cup T' \in \alpha(C_1)$ is established via V_1 and V_2, the immediate link between V_1 and V_2 leads to an immediate dependency between the variables in $(T_1^1 \cup T \cup T_2^1 \cup T') \setminus \{X\}$. So, $(T_1^1 \cup T \cup T_2^1 \cup T') \setminus \{X\} \in \alpha(C_1)$.

2. $c_1' \equiv (X = \mathcal{T}_1[V_1])$, $c_2' \equiv (V_2 = \mathcal{T}_2[X])$
   Analogous to case 1, but now applying rule *lu*2.
3. $c_1' \equiv (V_1 = \mathcal{T}_1[X])$, $c_2' \equiv (X = \mathcal{T}_2[V_2])$
   Analogous to case 2.
4. $c_1' \equiv (V_1 = \mathcal{T}_1[X])$, $c_2' \equiv (V_2 = \mathcal{T}_2[X])$
   Analogous to case 1, but now applying rule *li*1. □

The elimination of numerical variables is more complex. Two cases are distinguished : the elimination of a numerical term occurring in two constraints (via *lu*1, *lu*2, *ln*1 or *li*1), in Lemma 5.3.4, and the elimination of a set of numerical variables occurring in two or more constraints (via *li*2), in Lemma 5.3.5.

Lemma 5.3.4 (Elimination of a numerical term)
Let $C_1 \wedge C_2 \in SCons$. Assume that

1. *$(c_1; B_1)$ and $(c_2; B_2)$ are nodes in one labelled derivation tree or the roots of two labelled derivation trees over $C_1 \wedge C_2$;*
2. *e is a numerical term (special case : $e \equiv Y$ where Y is numerical in $C_1 \wedge C_2$) that occurs in both c_1 and c_2 with $vars(e) \subseteq B_1 \cap B_2$;*
3. *$(c_1; B_1)$ and $(c_2; B_2)$ are used as starting nodes in derivation rule *lu*1, *lu*2, *ln*1 or *li*1 to derive a node $(c_0; B_0)$ or an indirect dependency B_0 (note : $e \equiv Y$ in case of *lu*1, *lu*2 or *li*1); e is eliminated via this derivation step.*
4. *$B_1 = T_1^1 \cup T_1^2$ with $T_1^1 \cup T \in \alpha(C_1)$ and $T_1^2 \cup T \in \alpha(C_2)$,*
   $B_2 = T_2^1 \cup T_2^2$ with $T_2^1 \cup T' \in \alpha(C_1)$ and $T_2^2 \cup T' \in \alpha(C_2)$;
   (note : $B_0 = (B_1 \cup B_2) \setminus vars(e) = (T_1^1 \cup T_2^1 \cup T_1^2 \cup T_2^2) \setminus vars(e)$);
   $W^1 \subseteq vars(e)$ is the set of e-variables that occur only in $T_1^1 \cap T_2^1$ (not in $T_1^2 \cup T_2^2$);
   $W^2 \subseteq vars(e)$ is the set of e-variables that occur only in $T_1^2 \cap T_2^2$ (not in $T_1^1 \cup T_2^1$)

Let $B_0^1 = (T_1^1 \cup T_2^1) \cap B_0$ and $B_0^2 = (T_1^2 \cup T_2^2) \cap B_0$ (so $B_0^1 \cup B_0^2 = B_0$).
Let $E^1 = (T_1^1 \cup T_2^1) \cap vars(e)$ and $E^2 = (T_1^2 \cup T_2^2) \cap vars(e)$ (so $E^1 \cup E^2 = vars(e)$)[14].
So we can write
$T_1^1 \cup T_2^1 \cup T \cup T' = B_0^1 \uplus E^1 \uplus (T \cup T')$ and $T_1^2 \cup T_2^2 \cup T \cup T' = B_0^2 \uplus E^2 \uplus (T \cup T')$
($\uplus$ denotes disjoint union[15]).
Then
$(T_1^1 \cup T \cup T_2^1 \cup T') \setminus (W^1 \cup R^1 \cup R) = (B_0^1 \setminus R^1) \uplus [((E^1 \setminus W^1) \uplus (T \cup T')) \setminus R] \in \alpha(C_1)$,
$(T_1^2 \cup T \cup T_2^2 \cup T') \setminus (W^2 \cup R^2 \cup R) = (B_0^2 \setminus R^2) \uplus [((E^2 \setminus W^2) \uplus (T \cup T')) \setminus R] \in \alpha(C_2)$
where R^1, R^2 and R are sets of variables that are eliminated while eliminating W^1 and W^2. More precisely, $R^1 \subseteq B_0^1$, $R^2 \subseteq B_0^2$ and $R \subseteq (vars(e) \setminus (W^1 \cup W^2)) \uplus (T \cup T')$. Moreover, $(B_0^1 \setminus R^1) \cup (B_0^2 \setminus R^2) = B_0$, so no variables of B_0 are lost in the elimination process. Also note that $E^1 \setminus W^1 = E^2 \setminus W^2$.

[14] B_0^1 and E^1 (resp. B_0^2 and E^2) are the B_0 and E variables that occur within C_1 (resp. C_2); B_0^1 and B_0^2 (also E^1 and E^2) are not necessarily disjoint.

[15] B_0 and $vars(e)$ are disjoint with $T \cup T'$, since the latter is a subset of the variables that are eliminated while deriving $(c_1; B_1)$ and $(c_2; B_2)$ and those variables cannot be reintroduced.

The lemma is illustrated in Figure 5.13. The idea is that the C_1-links $T_1^1 \cup T$ and $T_2^1 \cup T'$ can be short-circuited to a new C_1-link via elimination of the variables in W^1 (i.e. the variables that occur only in the C_1-links and not in the C_2-links); this elimination causes simultaneous elimination of the sets of (numerical) variables R^1 and R. A similar short-circuiting is applied to the C_2-links $T_1^2 \cup T$ and $T_2^2 \cup T'$, via elimination of W^2 and thereby also of R^2 and R. Then $B_0^1 \setminus R^1$ and $B_0^2 \setminus R^2$ can be used resp. as S^1 and S^2, and $((E^1 \setminus W^1) \circlearrowleft (T \cup T')) \setminus R = ((E^2 \setminus W^2) \circlearrowleft (T \cup T')) \setminus R$ as S, such that $B_0 = S_1 \cup S_2$ with $S_1 \cup S \in \alpha(C_1)$ and $S_2 \cup S \in \alpha(C_2)$.
Before proving the lemma, we first give an example.

Example 5.3.7
Let $C_1 \equiv X - U = 3 \wedge Y - 2V = 0$ *and* $C_2 \equiv U - Z + V = 0 \wedge Y - Z + V = 1$; $sform(C_1 \wedge C_2) \equiv X - 2V = 2 \wedge Y - 2V = 0 \wedge U - 2V = -1 \wedge Z - 3V = -1$. *Consider* $X - 2V = 2$ *in* $sform(C_1 \wedge C_2)$; *a labelled derivation tree for* $(X - 2V = 2; \{X, V\})$ *is given in Figure 5.12.*

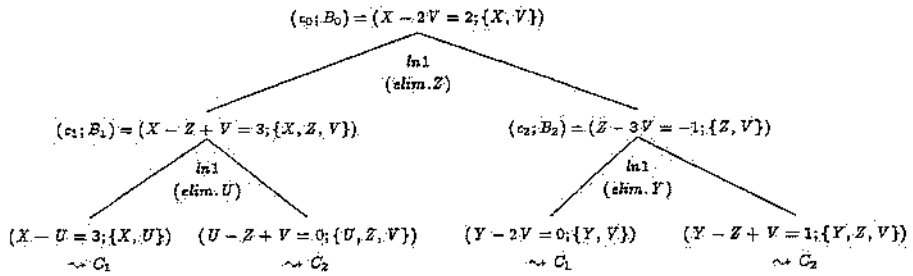


Figure 5.12: Labeled derivation tree for $(X - 2V = 2; \{X, V\})$

$B_0 = \{X, V\}$ *with* $B_0^1 = \{X, V\}$ *and* $B_0^2 = \{V\}$.
$B_1 = T_1^1 \cup T_1^2 = \{X, Z, V\}$ *with*
$T_1^1 \cup T = \{X\} \cup \{U\} \in \alpha(C_1)$ *and* $T_1^2 \cup T = \{V, Z\} \cup \{U\} \in \alpha(C_2)$.
$B_2 = T_2^1 \cup T_2^2 = \{Z, V\}$ *with*
$T_2^1 \cup T' = \{V\} \cup \{Y\} \in \alpha(C_1)$ *and* $T_2^2 \cup T' = \{V, Z\} \cup \{Y\} \in \alpha(C_2)$.
$vars(e) = \{Z\}$ *and* $E^1 = \emptyset$ *and* $E^2 = \{Z\}$. *There are no e-variables occurring only in the* C_1*-parts, so* $W^1 = \emptyset$; *there is one e-variable,* Z, *occurring only in the* C_2*-parts, so* $W^2 = \{Z\}$. *The variable* Z *must be eliminated by combining the* C_2*-constraints* $V - Z + U = 0$ *and* $V - Z + Y = 1$; *this causes simultaneous removal of* $V \in B_0^2$.
So, $R^1 = \emptyset$, $R = \emptyset$ *and* $R^2 = \{V\}$.
Then $B_0^1 \setminus R^1 = \{X, V\} \setminus \emptyset = \{X, V\}$; $B_0^2 \setminus R^2 = \{V\} \setminus \{V\} = \emptyset$ *and*
$((E^1 \setminus W^1) \circlearrowleft (T \cup T')) \setminus R = ((E^2 \setminus W^2) \circlearrowleft (T \cup T')) \setminus R = \{U, Y\}$.
So $(T_1^1 \cup T_2^1 \cup T \cup T') \setminus (W^1 \cup R^1 \cup R) = \{X, V\} \circlearrowleft \{U, Y\} \in \alpha(C_1)$ *and*
$(T_1^2 \cup T_2^2 \cup T \cup T') \setminus (W^2 \cup R^2 \cup R) = \emptyset \circlearrowleft \{U, Y\} \in \alpha(C_2)$.

PROOF (Lemma 5.3.4)
$(c_1; B_1)$ and $(c_2; B_2)$ are used as starting nodes in rule *lu*1, *lu*2, *li*1 or *ln*1 that causes the elimination of e.

- In case of *lu*1, *lu*2 and *li*1, e is a single variable Y and c_1 and c_2 both contain e.
- In case of *ln*1, c_1 contains $m * e$ and c_2 contains $n * e$ (where m and n are non-zero numbers).

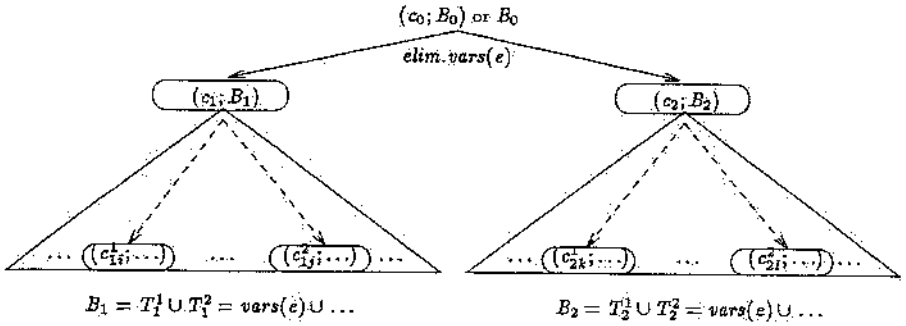


Figure 5.13: Situation given in Lemma 5.3.4.

By Lemma 5.3.2, $\mathbb{T}_{(c_1;B_1)}$ and $\mathbb{T}_{(c_2;B_2)}$ contain a set of leaves (related to C_1 and/or C_2) that contribute to e (or $m * e$ and $n * e$) in c_1 and c_2 :

- $\{(c^1_{11}; \ldots), \ldots, (c^1_{1m}; \ldots), (c^2_{11}; \ldots), \ldots, (c^2_{1n}; \ldots)\} \subseteq leaves(\mathbb{T}_{(c_1;B_1)})$ such that each c^k_{1i} is of the form $(e^k_{1i} \doteq d^k_{1i}$, with $vars(e^k_{1i}) \cap vars(d^k_{1i}) = \emptyset$, or $V^k_{1i} = T_{1i}[e^k_{1i}])$ and

$$e \doteq \overbrace{b^1_{11} e^1_{11} \dot{+} \ldots \dot{+} b^1_{1m} e^1_{1m}}^{e^1_1 \doteq} \dot{+} \overbrace{b^2_{11} e^2_{11} \dot{+} \ldots \dot{+} b^2_{1n} e^2_{1n}}^{e^2_1 \doteq} \tag{5.1}$$

  with $vars(e^k_{1i}) \subseteq vars(e)$[16]. All c^1_{1i} belong to C_1, all c^2_{1i} belong to C_2. So e is obtained by summing up subexpressions in constraints of C_1 and C_2, where we let e^1_1 be the normal form of the sum of the C_1-expressions and e^2_1 be the normal form of the sum of the C_2-expressions; $vars(e^1_1) \subseteq T^1_1$ and $vars(e^2_1) \subseteq T^2_1$. Note that $vars(e^1_1) \cap vars(e^2_1)$ is not necessarily empty.
- $\{(c^1_{21}; \ldots), \ldots, (c^1_{2p}; \ldots), (c^2_{21}; \ldots), \ldots, (c^2_{2q}; \ldots)\} \subseteq leaves(\mathbb{T}_{(c_2;B_2)})$ such that each c^k_{2j} is of the form $(e^k_{2j} \doteq d^k_{2j}$, with $vars(e^k_{2j}) \cap vars(d^k_{2j}) = \emptyset$, or $V^k_{2j} = T_{ij}[e^k_{2j}])$ and

$$e \doteq \overbrace{b^1_{21} e^1_{21} \dot{+} \ldots \dot{+} b^1_{2p} e^1_{2p}}^{e^1_2 \doteq} \dot{+} \overbrace{b^2_{21} e^2_{21} \dot{+} \ldots \dot{+} b^2_{2q} e^2_{2q}}^{e^2_2 \doteq} \tag{5.2}$$

  with $vars(e^k_{2j}) \subseteq vars(e)$[17]. All c^1_{2j} belong to C_1, all c^2_{2j} belong to C_2. So $vars(e^1_2) \subseteq T^1_2$ and $vars(e^2_2) \subseteq T^2_2$. Note that $vars(e^1_2) \cap vars(e^2_2)$ is not necessarily empty.

The variables in W^1 (resp. W^2) can only occur in the C_1-constraints c^1_{1i} and c^1_{2j} (resp. the C_1-constraints c^2_{1i} and c^2_{2j}), since it is given that they occur only in T^1_1 and T^1_2 (resp. T^2_1 and T^2_2). The C_1-link between W^1 and the other variables in $T^1_1 \cup T$ (resp. $T^1_2 \cup T'$) is established via the d^1_{1i} and V^1_{1i} (resp. the d^1_{2j} and V^1_{2j}). Similarly, the C_2-link between W^2 and the other variables in $T^2_1 \cup T$ (resp. $T^2_2 \cup T'$) is established via the d^2_{1i} and V^2_{1j} (resp. the d^2_{2i} and V^2_{2j}). Referring to e^1_1, e^2_1, e^1_2 and e^2_2, the sets W^1 and W^2 can be characterised more precisely: $W^1 = (vars(e^1_1) \cap vars(e^1_2)) \setminus (vars(e^2_1, e^2_2))$ and $W^2 =$

[16]The factor m, in case of $m * e$ in c_1, is absorbed in the coefficients b^k_{1i}.
[17]The factor n, in case of $n * e$ in c_2, is absorbed in the coefficients b^k_{2j}.

$(vars(e_1^2) \cap vars(e_2^2)) \setminus (vars(e_1^1, e_2^1))$. Figure 5.14 shows the different kinds of variables in e. Notice that (1) $vars(e_1^1, e_2^1) = E^1$, $vars(e_1^2, e_2^2) = E^2$, and (2) $E^1 \setminus W^1 = E^2 \setminus W^2$.

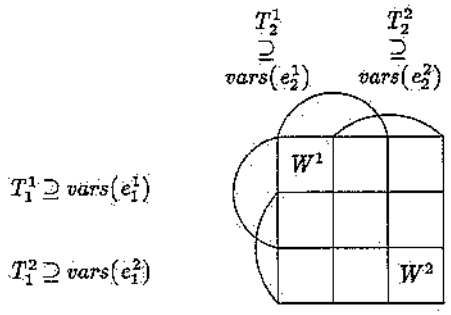


Figure 5.14: Splitting up the variables in $vars(e)$

It is now shown that the combination of the C_1-constraints c_{1i}^1 and c_{2j}^1 allows to short-circuit the given C_1-links ($T_1^1 \cup T$ and $T_2^1 \cup T'$) via elimination of the variables in W^1.

Assume $\mathcal{N}_1^1 \subseteq \{c_{11}^1, \ldots, c_{1m}^1\}$ is the set of numerical constraints c_{1i}^1 (of the form $e_{1i}^1 = d_{1i}^1$) and $\mathcal{U}_1^1 \subseteq \{c_{11}^1, \ldots, c_{1m}^1\}$ is the set of unification constraints c_{1i}^1 (of the form $V_{1i}^1 = \mathcal{T}_{1i}[e_{1i}^1]$). Similarly, $\mathcal{N}_2^1 \subseteq \{c_{21}^1, \ldots, c_{2p}^1\}$ is the set of numerical constraints c_{2j}^1 (of the form $e_{2j}^1 = d_{2j}^1$) and $\mathcal{U}_2^1 \subseteq \{c_{21}^1, \ldots, c_{2p}^1\}$ is the set of unification constraints c_{2j}^1 (of the form $V_{2j}^1 = \mathcal{T}_{1j}[e_{2j}^1]$). So $\{c_{11}^1, \ldots, c_{1m}^1\}$ is split into $\mathcal{N}_1^1 \dot\cup \mathcal{U}_1^1$ and $\{c_{21}^1, \ldots, c_{2p}^1\}$ into $\mathcal{N}_2^1 \dot\cup \mathcal{U}_2^1$ where $\dot\cup$ denotes disjoint union[18].

The elimination of the variables in W^1 in general includes three cases.

1. **Elimination of the $Y_r \in W^1$ that occur only in $\mathcal{N}_1^1 \cup \mathcal{N}_2^1$ (not in $\mathcal{U}_1^1 \cup \mathcal{U}_2^1$)** Consider the following linear combination of the constraints $e_{1i}^1 = d_{1i}^1$ and $e_{1j}^1 = d_{1j}^1$ in $\mathcal{N}_1^1 \cup \mathcal{N}_2^1$:

$$\overbrace{\dot\sum_{\mathcal{N}_1^1} b_{1i}^1 e_{1i}^1}^{esum_1^1} \dot- \overbrace{\dot\sum_{\mathcal{N}_2^1} b_{2j}^1 e_{2j}^1}^{esum_2^1} = \overbrace{\dot\sum_{\mathcal{N}_1^1} b_{1i}^1 d_{1i}^1}^{dsum_1^1} \dot- \overbrace{\dot\sum_{\mathcal{N}_2^1} b_{2j}^1 d_{2j}^1}^{dsum_2^1} \tag{5.3}$$

(Note that $esum_1^1$ resp. $esum_2^1$ is a part of e_1^1 resp. e_2^1.) This equation belongs to C_1^* (and its abstraction to $\alpha(C_1)$) since all constraints in $\mathcal{N}_1^1$ and $\mathcal{N}_2^1$ belong to C_1. For each $Y_r \in W^1$ (so $Y_r \notin e_1^2$ and hence $coef_{Y_r}(e_1^2) = 0$), we know via (5.1) that $coef_{Y_r}(e) = coef_{Y_r}(e_1^1)$ where $coef_X(exp)$ denotes the coefficient of X in the normal form of the expression exp. If Y_r occurs only in $\mathcal{N}_1^1 \cup \mathcal{N}_2^1$ (not in $\mathcal{U}_1^1 \cup \mathcal{U}_2^1$), then $coef_{Y_r}(esum_1^1) = coef_{Y_r}(e_1^1)$. Combining the above we have that $coef_{Y_r}(esum_1^1) = coef_{Y_r}(e)$. Similarly, $coef_{Y_r}(e) = coef_{Y_r}(e_2^1)$ via (5.2) and $coef_{Y_r}(e_2^1) = coef_{Y_r}(esum_2^1)$. So, $coef_{Y_r}(esum_1^1) = coef_{Y_r}(esum_2^1)$ and all these Y_r are eliminated *simultaneously* via (5.3).

2. **Elimination of the $Y_r \in W^1$ that occur only in $\mathcal{U}_1^1 \cup \mathcal{U}_2^1$ (not in $\mathcal{N}_1^1 \cup \mathcal{N}_2^1$)** Each $Y_r \in W^1$ that occurs only in $\mathcal{U}_1^1 \cup \mathcal{U}_2^1$ can be eliminated by pairwise combining the constraints of the form $V_{1i}^1 = \mathcal{T}_{1i}[Y_r]$ or $V_{2j}^1 = \mathcal{T}_{2j}[Y_r]$ (applying derivation rule *li*1). This

[18] We choose to put a constraint of the form $X = Y$ or $X = n$ (with n a number) into $\mathcal{N}$ and not into $\mathcal{U}$; so, $\mathcal{U}$ contains only "real" unification constraints of the form $X = \mathcal{T}^+[\ldots]$ (a compound Herbrand term) or $X = f$ (f is a non-numerical constant).

results in dependencies between two of the V^1_{1i}/V^1_{2j}. Each of these dependencies belongs to $\alpha(C_1)$ since $\mathcal{U}^1_1$ and $\mathcal{U}^1_2$ are subsets of C_1.

3. **Elimination of the $Y_r \in W^1$ that occur both in $\mathcal{U}^1_1 \cup \mathcal{U}^1_2$ and in $\mathcal{N}^1_1 \cup \mathcal{N}^1_2$.** For each such an Y_r, let $\mathcal{Z}_r$ be the subset of constraints of $\mathcal{U}^1_1 \cup \mathcal{U}^1_2$ in which Y_r occurs. All the Y_r can be eliminated *simultaneously* by applying derivation rule *li2* several times : each time, a constraint is taken from each $\mathcal{Z}_r$ and combined with (5.3); this yields a dependency between the d^1_{1i}/d^1_{2j} and some of the V^1_{1i}/V^1_{2j} (according to the constraint chosen from $\mathcal{Z}_r$). This dependency belongs to $\alpha(C_1)$ since all of the involved constraints belong to C_1.

Taking the union of all the dependencies derived in cases 1, 2 and 3 above results in a dependency between all d^1_{1i}/d^1_{2j} and V^1_{1i}/V_{2j} in which the W^1 variables do no longer occur. This dependency belongs to $\alpha(C_1)$ since all dependencies in the union belong to $\alpha(C_1)$ and $\alpha(C_1)$ is closed under union. Recall that the link between the variables in W^1 and the other variables in $T^1_1 \cup T$ (resp. $T^1_2 \cup T'$) is established via the d^1_{1i}/d^1_{2j} and V^1_{1i}/V^1_{2j}. Hence, there is also a dependency between the variables in $(T^1_1 \cup T \cup T^1_2 \cup T') \setminus W^1$. However, this is not entirely correct : via (5.3), a variable X in $T^1_1 \cup T \cup T^1_2 \cup T'$ is also eliminated from this set (besides the W^1 variables) (1) if it occurs only in $\mathcal{N}^1_1 \cup \mathcal{N}^1_2$ and in no other C_1-constraints used in the derivation of $T^1_1 \cup T \cup T^1_2 \cup T'$ and (2) if it has the same coefficient in $esum^1_1$ and in $esum^1_2$, or in $dsum^1_1$ and $dsum^1_2$. A distinction is made between variables that are eliminated from B^1_0, say the set R^1, and those that are eliminated from $(E^1 \setminus W^1) \dot{\cup} (T \cup T')$, say the set R. The result then becomes $(B^1_0 \setminus R^1) \dot{\cup} [((E^1 \setminus W^1) \dot{\cup} (T \cup T')) \setminus R] \in \alpha(C_1)$.

In a similar way to the elimination of the W^1 variables, the W^2 variables can be eliminated using the c^2_{1k} and c^2_{2l} constraints in C_2, thereby short-circuiting the $T^2_1 \cup T$ and $T^2_2 \cup T'$ links into a new C_2 dependency. Again, elimination of W^2 may cause the elimination of an additional set of variables R^2 from B^2_0 and R from $(E^2 \setminus W^2) \dot{\cup} (T \cup T')$. The result then becomes $(B^2_0 \setminus R^2) \dot{\cup} [((E^2 \setminus W^2) \dot{\cup} (T \cup T')) \setminus R] \in \alpha(C_2)$.

It still has to be shown that (1) $(B^1_0 \setminus R^1) \cup (B^2_0 \setminus R^2) = B_0$, i.e. no B_0 variables are lost; (2) the same set R can be subtracted from both $(E^1 \setminus W^1) \dot{\cup} (T \cup T')$ and $(E^2 \setminus W^2) \dot{\cup} (T \cup T')$, i.e. if a variable in $vars(e) \setminus (W^1 \cup W^2)$ or $T \cup T'$ is eliminated from the C_1-side during the elimination of W^1 (resp. from the C_2-side during the elimination of W^2) then it can also be eliminated from the C_2-side (resp. the C_1-side).

1. To prove that $(B^1_0 \setminus R^1) \cup (B^2_0 \setminus R^2) = B_0$, three things must be shown :
   a. **If $X \in B^1_0$ and $X \notin B^2_0$, then $X \notin R^1$**
   (Proof by refutation) Suppose $X \in R^1$, i.e. X is eliminated due to the elimination of W^1 on the C_1-side. This implies that $coef_X(dsum^1_1) = coef_X(dsum^1_2)$ and $dsum^1_1$ and $dsum^1_2$ contain *all* occurrences of X on C_1-side (i.e. X occurs nowhere else in C_1-constraints). Since $X \notin B^2_0$, i.e. there is no contribution to X from C_2-side, we have that $coef_X(dsum^1_1) = coeff_X(c_1)$ and $coef_X(dsum^1_2) = coef_X(c_2)$; since $coef_X(dsum^1_1) = coef_X(dsum^1_2)$ (cf. above), also $coeff_X(c_1) = coef_X(c_2)$. This implies that X would no longer be in B_0 but would be eliminated in the derivation step from $(c_1; B_1)$ and $(c_2; B_2)$ to B_0. This gives a contradiction with $X \in B^1_0 \subseteq B_0$. So, the assumption $X \in R^1$ is wrong.
   b. **If $X \in B^2_0$ and $X \notin B^1_0$, then $X \notin R^2$**
   similar

c. If $X \in B_0^1 \cap B_0^2$, then X not both in R^1 and R^2
(Proof by refutation) Suppose that $X \in R^1$ and $X \in R^2$. By $X \in R^1$, we have that $coef_X(dsum_1^1) = coef_X(dsum_2^1)$ and $dsum_1^1$ and $dsum_2^1$ contain *all* occurrences of X on C_1-side. Similarly, by $X \in R^2$, we have that[19] $coef_X(dsum_1^2) = coef_X(dsum_2^2)$ and $dsum_1^2$ and $dsum_2^2$ contain *all* occurrences of X on the C_2-side. So, $coef_X(dsum_1^1 \dot{+} dsum_2^1) = coef_X(c_1) = coef_X(dsum_2^1 \dot{+} dsum_2^2) = coef_X(c_2)$. This implies that X would no longer be in B_0 but would be eliminated in the derivation step from $(c_1; B_1)$ and $(c_2; B_2)$ to B_0. This gives a contradiction with $X \in B_0^1 \cap B_0^2 \subseteq B_0$. So, the assumption $X \in R^1 \cap R^2$ is wrong.

2. Finally we have to prove that, if $X \in (E^1 \setminus W^1) \dot{\cup} (T \cup T') = (E^2 \setminus W^2) \dot{\cup} (T \cup T')$ and X is eliminated on the C_1-side while eliminating W^1, then X can also be eliminated on the C_2-side while eliminating W^2. First, consider $X \in (E^1 \setminus W^1) = (E^2 \setminus W^2)$. Since $X \in vars(e)$, we obtain via (5.1) and (5.2) that
$$coef_X(e_1^1 \dot{+} e_1^2) = coef_X(e_2^1 \dot{+} e_2^2) = coef_X(e). \tag{5.4}$$
Given that X is eliminated on the C_1-side and therefore the only occurrences of X on C_1-side are the ones in $esum_1^1$ and $esum_2^1$ (i.e. in $\mathcal{N}_1^1 \cup \mathcal{N}_2^1$), we know that $coef_X(esum_1^1) = coef_X(esum_2^1) = coef_X(e_1^1) = coef_X(e_2^1)$. Combining this with 5.4 gives $coef_X(e_1^2) = coef_X(e_2^2)$ and therefore X can also be eliminated on the C_2-side.
Secondly, consider $X \in T \cup T'$. Suppose that X is eliminated on the C_1-side, then
$$coef_X(dsum_1^1) = coef_X(dsum_2^1) \tag{5.5}$$
and X occurs only in $\mathcal{N}_1^1 \cup \mathcal{N}_2^1$ on the C_1-side. Let $Xsum_1^2$ be the sum of all subterms (each multiplied by some factor) in the constraints of C_2 that involve X and that are in $leaves(\mathbb{T}_{(c_1;B_1)})$ (note : $Xsum_1^2$ is similar to $esum_1^2$ but now contributing to X instead of e). Then
$$coef_X(c_1) = coef_X(dsum_1^1 \dot{+} Xsum_1^2). \tag{5.6}$$
Similarly, let $Xsum_2^2$ be the sum of contributions to X in the constraints of C_2 that are in $leaves(\mathbb{T}_{(c_2;B_2)})$ ($Xsum_2^2$ is similar to $esum_2^2$ but now contributing to X instead of e). Then
$$coef_X(c_2) = coef_X(dsum_2^1 \dot{+} Xsum_2^2). \tag{5.7}$$
Since X does not occur in c_1 nor c_2, we have that $coef_X(c_1) = coef_X(c_2) = 0$. Combining this with (5.5), (5.6) and (5.7) yields $coef_X(Xsum_1^2) = coef_X(Xsum_2^2)$ and hence X can also be eliminated on the C_2-side. □

Lemma 5.3.5 (Elimination of a set of numerical variables)
Let $C_1 \wedge C_2 \in SCons$.

1. *$(c_0; B_0), (c_1; B_1), \ldots, (c_m; B_m)$ are the roots of labelled derivation trees over $C_1 \wedge C_2$;*
2. *e is a numerical term of the form $a_1 Y_1 + \ldots + a_m Y_m$ that occurs in c_0 and $vars(e) \subseteq B_0$; each Y_i also occurs in c_i and B_i;*
3. *$(c_0; B_0), (c_1; B_1), \ldots, (c_m; B_m)$ are used as starting nodes in li2 to derive an indirect dependency D; $\{Y_1, \ldots, Y_m\}$ are eliminated via this derivation step. More precisely, c_0 is of the form $a_1 Y_1 + \ldots + a_m Y_m = \ldots$ and each c_i is of the form $X_i = T_i^+[Y_i]$.*
4. *$B_0 = T_0^1 \cup T_0^2$ with $T_0^1 \cup T_0 \in \alpha(C_1)$ and $T_0^2 \cup T_0 \in \alpha(C_2)$,*
   $B_i = T_i^1 \cup T_i^2$ with $T_i^1 \cup T_i \in \alpha(C_1)$ and $T_i^2 \cup T_i \in \alpha(C_2)$;

[19] $dsum_1^2$ and $dsum_2^2$ are the combinations of subterms in the right-hand sides of the C_2-constraints contributing to e and are comparable to $dsum_1^1$ and $dsum_2^1$ on the C_1-side.

The variables in $W^1 = \{Y_1, \ldots, Y_k\}$ $(0 \leq k \leq m)$ *occur* only *in the* C_1*-parts (i.e. in* $T_0^1 \cup T_0 \cup \ldots \cup T_m^1 \cup T_m$*) and not in the* C_2*-parts,*
The variables in $W^2 = \{Y_{k+1}, \ldots, Y_l\}$ $(k \leq l \leq m)$ *occur* only *in the* C_2*-parts (i.e. in* $T_0^2 \cup T_0 \cup \ldots \cup T_m^2 \cup T_m$*) and not in the* C_1*-parts*

Let $D^1 = D \cap (T_0^1 \cup \ldots \cup T_m^1)$ *and* $D^2 = D \cap (T_0^2 \cup \ldots \cup T_m^2)$ *(so* $D^1 \cup D^2 = D$*).*
Let $E^1 = vars(e) \cap (T_0^1 \cup \ldots \cup T_m^1)$ *and* $E^2 = vars(e) \cap (T_0^2 \cup \ldots \cup T_m^2)$ *(so* $E^1 \cup E^2 = vars(e)$*).*
Then
$(T_0^1 \cup T_0 \cup \ldots \cup T_m^1 \cup T_m) \setminus (W^1 \cup R) = D^1 \dot{\cup} [((E^1 \setminus W^1) \cup (T_0 \cup \ldots \cup T_m)) \setminus R] \in \alpha(C_1)$,
$(T_0^2 \cup T_0 \cup \ldots \cup T_m^2 \cup T_m) \setminus (W^2 \cup R) = D^2 \dot{\cup} [((E^2 \setminus W^2) \cup (T_0 \cup \ldots \cup T_m)) \setminus R] \in \alpha(C_2)$
where R *is the set of variables that is eliminated while eliminating* W^1 *and* W^2*. More precisely,* $R \subseteq (vars(e) \setminus (W^1 \cup W^2)) \dot{\cup} (T_0 \cup \ldots \cup T_m)$*. Note that* $E^1 \setminus W^1 = E^2 \setminus W^2$*.*

The idea is that the C_1-links $T_0^1 \cup T_0, \ldots, T_m^1 \cup T_m$ can be short-circuited to a new C_1-link via elimination of the variables in W^1 (that occur only in the C_1-links); this elimination causes simultaneous elimination of the set of variables R. A similar short-circuiting is applied to the C_2-links $T_0^2 \cup T_0, \ldots, T_m^2 \cup T_m$, via elimination of W^2 and thereby also of R. No variables of D can disappear in the elimination process.

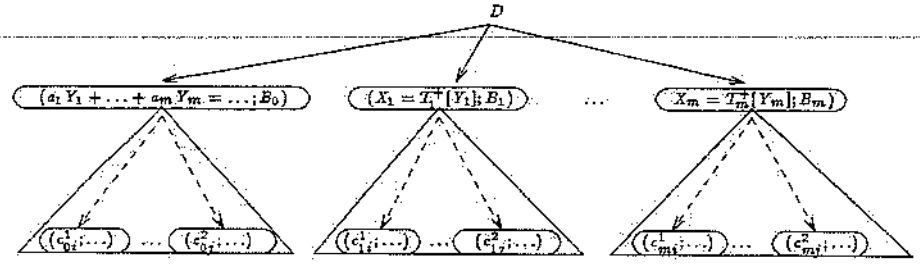


Figure 5.15: Situation given in Lemma 5.3.5.

PROOF
The proof is very similar to that of Lemma 5.3.4.

$(c_0; B_0)$ contains $e \equiv a_1 Y_1 + \ldots + a_m Y_m$. By Lemma 5.3.2, there exists $\{(c_{01}^1; \ldots), \ldots, (c_{0p_0}^1; \ldots), (c_{01}^2; \ldots), \ldots, (c_{0q_0}^2; \ldots)\} \subseteq leaves(\mathbb{T}_{(c_0; B_0)})$ such that each c_{0i}^k is of the form $(e_{0i}^k = d_{0i}^k$ or $V_{0i}^k = \mathcal{T}_{0i}[e_{0i}^k])$ and

$$e \doteq \overbrace{b_{01}^1 e_{01}^1 \dot{+} \ldots \dot{+} b_{0p_0}^1 e_{0p_0}^1}^{e_0^1 \doteq} \dot{+} \overbrace{b_{01}^2 e_{01}^2 \dot{+} \ldots \dot{+} b_{0q_0}^2 e_{0q_0}^2}^{e_0^2 \doteq} \tag{5.8}$$

with $vars(e_{0i}^k) \subseteq vars(e)$. All c_{0i}^1 belong to C_1, all c_{0i}^2 belong to C_2. So e_0^1 is the normal form of the sum of the C_1-expressions and e_0^2 is the normal form of the sum of the C_2-expressions that contribute to e; $vars(e_0^1) \subseteq T_0^1$ and $vars(e_0^2) \subseteq T_0^2$. Note that $vars(e_0^1) \cap vars(e_0^2)$ is not necessarily empty.

We also consider the occurrences of $\{Y_1, \ldots, Y_m\}$ in the leaves of $\mathbb{T}_{(c_1; B_1)}, \ldots, \mathbb{T}_{(c_m; B_m)}$ (note: although Y_i occurs only in B_0 and B_i, it may still be involved in the derivation

of the other B_j, where it is eliminated in some intermediate derivation step). This is done via the extension of Lemma 5.3.2 : it states that there exist $\{(c^1_{i1};\ldots),\ldots,(c^1_{ip_i};\ldots),(c^2_{i1};\ldots),\ldots,(c^2_{iq_i};\ldots)\}\subseteq leaves(\mathbb{T}_{(c_2;B_2)})$ such that each c^k_{ij} is of the form ($e^k_{ij} = d^k_{ij}$ or $V^k_{ij} = T_{ij}[e^k_{ij}]$) and

$$Y_i \doteq \overbrace{b^1_{i1}e^1_{i1} \dot{+} \ldots \dot{+} b^1_{ip_i}e^1_{ip_i}}^{e^1_i \doteq} \dot{+} \overbrace{b^2_{i1}e^2_{i1} \dot{+} \ldots \dot{+} b^2_{iq_i}e^2_{iq_i}}^{e^2_i \doteq} \tag{5.9}$$

with $vars(e^k_{ij})\subseteq\{Y_1,\ldots,Y_m\}$. Note that $coef_{Y_r}(e^1_i \dot{+} e^2_i)$ is 0 for $r \neq i$ and 1 for $r = i$. All c^1_{ij} belong to C_1, all c^2_{ij} belong to C_2. So e^1_i and e^2_i are resp. the C_1- and C_2-contributions to $Y_1,\ldots,Y_m$ in $\mathbb{T}_{(c_i;B_i)}$; $vars(e^1_i)\subseteq T^1_i$ and $vars(e^2_i)\subseteq T^2_i$.

The variables in W^1 (resp. W^2) can only occur in the c^1_{ij} (resp. the c^2_{ij}). The C_1-link between W^1 and the other variables in $T^1_i \cup T_i$ is established via the d^1_{ij} and V^1_{ij}. Similarly, the C_2-link between W^2 and the other variables in $T^2_i \cup T_i$ is established via the d^2_{ij} and V^2_{ij}.

It has to be shown that the combination of the c^1_{ij} allows to short-circuit the given C_1-links $(T^1_1 \cup T_1,\ldots,T^1_m \cup T_m)$ via elimination of the variables in W^1.

Assume that $\mathcal{N}^1_i \subseteq \{c^1_{i1},\ldots,c^1_{ip_i}\}$ is the set of numerical constraints c^1_{ij} (of the form $e^1_{ij} = d^1_{ij}$) and $\mathcal{U}^1_i \subseteq \{c^1_{i1},\ldots,c^1_{ip_i}\}$ is the set of unification constraints c^1_{ij} (of the form $V^1_{ij} = T_{ij}[e^1_{ij}]$) for each i in $0..m$. So $\{c^1_{i1},\ldots,c^1_{ip_i}\} = \mathcal{N}^1_i \uplus \mathcal{U}^1_i$.

The elimination of the variables in W^1 in general includes three cases.

1. **Elimination of the $Y_r \in W^1$ that occur only in $\mathcal{N}^1_0 \cup \ldots \cup \mathcal{N}^1_m$ (not in $\mathcal{U}^1_0 \cup \ldots \cup \mathcal{U}^1_m$).**

Consider the following linear combination of the constraints $e^1_{ij} = d^1_{ij}$ in the $\mathcal{N}^1_i$:

$$\begin{aligned} &\overbrace{\textstyle\sum_{\mathcal{N}^1_0} b^1_{0j}e^1_{0j}}^{esum^1_0} \dot{-} a_1\overbrace{\textstyle\sum_{\mathcal{N}^1_1} b^1_{1j}e^1_{1j}}^{esum^1_1} \dot{-} \ldots \dot{-} a_m\overbrace{\textstyle\sum_{\mathcal{N}^1_m} b^1_{mj}e^1_{mj}}^{esum^1_m} \\ = &\underbrace{\textstyle\sum_{\mathcal{N}^1_0} b^1_{0j}d^1_{0j}}_{dsum^1_0} \dot{-} a_1\underbrace{\textstyle\sum_{\mathcal{N}^1_1} b^1_{1j}d^1_{1j}}_{dsum^1_1} \dot{-} \ldots \dot{-} a_m\underbrace{\textstyle\sum_{\mathcal{N}^1_m} b^1_{mj}d^1_{mj}}_{dsum^1_m} \end{aligned} \tag{5.10}$$

This equation belongs to C^*_1 since all constraints in the $\mathcal{N}^1_i$ belong to C_1. For each $Y_r \in W^1$ (so $Y_r \notin e^2_i$ for i in $0..m$), we know via (5.8) that $coef_{Y_r}(e) = coef_{Y_r}(e^1_0) = a_r$. If Y_r occurs only in $\mathcal{N}^1_0$ (not in $\mathcal{U}^1_0$), then $coef_{Y_r}(esum^1_0) = coef_{Y_r}(e^1_0) = a_r$. Similarly, since $Y_r \in W^1$ and only in $\mathcal{N}^1_r$ (not in $\mathcal{U}^1_r$) we know via (5.9) that $coef_{Y_r}(esum^1_r) = 1$. Also, since $Y_r \in W^1$ and only in $\mathcal{N}^1_i$ (not in $\mathcal{U}^1_i$) we know that $coef_{Y_r}(esum^1_i) = 0$ for each i in $0..m$ with $i \notin \{0,r\}$. So all these Y_r are eliminated simultaneously via (5.10).

2. **Elimination of the $Y_r \in W^1$ that occur only in $\mathcal{U}^1_0 \cup \ldots \cup \mathcal{U}^1_m$ (not in $\mathcal{N}^1_0 \cup \ldots \cup \mathcal{N}^1_m$).**

Each $Y_r \in W^1$ that occurs only in $\mathcal{U}^1_0 \cup \ldots \cup \mathcal{U}^1_m$ can be eliminated by pairwise combining the constraints of the form $V^1_{ij} = T_{ij}[Y_r]$ (applying derivation rule $\mathcal{U}1$). This results in dependencies between two of the V^1_{ij}. Each of these dependencies belongs to $\alpha(C_1)$ since $\mathcal{U}^1_0,\ldots,\mathcal{U}^1_m$ are subsets of C_1.

3. Elimination of the $Y_r \in W^1$ that occur both in $\mathcal{U}_0^1 \cup \ldots \cup \mathcal{U}_m^1$ and in $\mathcal{N}_0^1 \cup \ldots \cup \mathcal{N}_m^1$.

For each such an Y_r, let $\mathcal{Z}_r$ be the subset of constraints in $\mathcal{U}_0^1 \cup \ldots \cup \mathcal{U}_m^1$ in which Y_r occurs. All the Y_r can be eliminated *simultaneously* by applying derivation rule *li2* several times. Each time, a constraint is taken from each $\mathcal{Z}_r$ and combined with (5.10); this yields a dependency between the d_{ij}^1 and some of the V_{ij}^1 (according to the constraint chosen from $\mathcal{Z}_r$). This dependency belongs to $\alpha(C_1)$ since all of the involved constraints belong to C_1.

Taking the union of all the dependencies derived in cases 1, 2 and 3 above results in a dependency between all d_{ij}^1 and V_{ij}^1 in which the W^1 variables do no longer occur. This dependency belongs to $\alpha(C_1)$ since $\alpha(C_1)$ is closed. Recall that the links between the variables in W^1 and the other variables in the $T_i^1 \cup T_i$ is established via the d_{ij}^1 and V_{ij}^1. Hence, there is also a dependency between the variables in $(T_0^1 \cup T_0 \cup \ldots \cup T_m^1 \cup T_m) \setminus W^1$. However, this is not entirely correct: via (5.10), a variable X in $T_0^1 \cup T_0 \cup \ldots \cup T_m^1 \cup T_m$ is also eliminated on the C_1-side (besides the W^1 variables) (1) if it occurs only in the $\mathcal{N}_i^1$ and in no other constraints used in the derivation of $T_0^1 \cup T_0 \cup \ldots \cup T_m^1 \cup T_m$ and (2) if it has the same coefficient in $esum_0^1$ and in $\sum_{i=1}^m a_i esum_i^1$, or in $dsum_0^1$ and $\sum_{i=1}^m a_i dsum_i^1$. No variables can be eliminated from $D_0 = (T_0^1 \cup \ldots \cup T_m^1) \setminus vars(e)$ (cf. below). The set of variables R that is eliminated besides W^1 is therefore a subset of $(E^1 \setminus W^1) \dot{\cup} (T_0 \cup \ldots \cup T_m)$. The result then becomes $D^1 \dot{\cup} [((E^1 \setminus W^1) \dot{\cup} (T_0 \cup \ldots \cup T_m)) \setminus R] \in \alpha(C_1)$.

A similar reasoning allows to show that the W^2 variables can be eliminated using the c_{ij}^2 constraints in C_2, thereby short-circuiting the $T_i^2 \cup T_i$ links into a new C_2-dependency. Again, elimination of W^2 may cause the elimination of an additional set of variables R from $(E^2 \setminus W^2) \dot{\cup} (T_0 \cup \ldots \cup T_m)$. The result then is $D^2 \dot{\cup} [((E^2 \setminus W^2) \dot{\cup} (T_0 \cup \ldots \cup T_m)) \setminus R] \in \alpha(C_2)$.

It still has to be shown that (1) no variables of D can be eliminated while eliminating W^1 and W^2; (2) the same set R can be subtracted from both $(E^1 \setminus W^1) \dot{\cup} (T_0 \cup \ldots \cup T_m)$ and $(E^2 \setminus W^2) \dot{\cup} (T_0 \cup \ldots \cup T_m)$.

1. The leaves in $\mathbb{T}_{(c_0;B_0)}, \ldots, \mathbb{T}_{(c_n;B_n)}$ contribute to $B_0 \setminus vars(e)$ and $B_i \setminus \{Y_i\} = X_i$ $(1 \leq i \leq m)$ in D. For each of these variables, it must be shown that it cannot be eliminated from D while eliminating W^1 and W^2. First of all, a X_i cannot be eliminated via (5.10) since X_i is non-numerical in $C_1 \wedge C_2$ and non-numerical variables may not appear in the numerical constraint (5.10). Secondly, a variable $Z \in B_0 \setminus vars(e)$ cannot be eliminated during elimination of W^1 via (5.10), since Z cannot occur in any of the N_i^1. This is shown by refutation. Suppose Z occurs in some of the $\mathcal{N}_i^1$. Since $Z \notin B_i$, this would imply that Z is eliminated during the derivation of B_i. However, Z occurs in D which would mean that an eliminated variable is reintroduced. Since this is not possible, the assumption that Z can be eliminated via (5.10) is wrong. Similarly Z cannot be eliminated during elimination of W^2.

2. Finally we have to prove that, if $V \in (E^1 \setminus W^1) \dot{\cup} (T_0 \cup \ldots \cup T_m) = (E^2 \setminus W^2) \dot{\cup} (T_0 \cup \ldots \cup T_m)$ and V is eliminated on the C_1-side while eliminating W^1, then V can also be eliminated on the C_2-side while eliminating W^2. First, consider $V = Y_i$ in $(E^1 \setminus W^1) = (E^2 \setminus W^2)$. From (5.8), we know that

$$coef_{Y_i}(e_0^1 \dot{+} e_0^2) = a_i, \; coef_{Y_i}(e_i^1 \dot{+} e_i^2) = 1,$$

$$coef_{Y_i}(e_j^1 \dot{+} e_j^2) = 0 \text{ for } j \text{ in } \{1,\ldots,m\} \setminus \{i\}$$

This implies

$$coef_{Y_i}((e_0^1 \dot{+} e_0^2) \dot{-} \dot{\sum}_{i=1}^m a_i(e_i^1 \dot{+} e_i^2)) = 0. \tag{5.11}$$

Given that Y_i is eliminated on the C_1-side while eliminating W^1 we have that $coef_{Y_i}(esum_0^1 \dot{-} \dot{\sum}_{i=1}^m a_i esum_i^1) = 0$; moreover, $coef_{Y_i}(esum_j^1) = coef_{Y_i}(e_j^1)$ for all j in $\{1,\ldots,m\}$ since Y_i is known to occur only in $\mathcal{N}_0^1 \cup \ldots \cup \mathcal{N}_m^1$ on the C_1-side. Hence

$$coef_{Y_i}(e_0^1 \dot{-} \dot{\sum}_{i=1}^m a_i e_i^1) = 0 \tag{5.12}$$

Subtracting (5.12) from (5.11) yields $coef_{Y_i}(e_0^2 \dot{-} \dot{\sum}_{i=1}^m a_i e_i^2) = 0$. So, Y_i can also be eliminated on C_2-side.

Secondly, consider $V \in (T_0 \cup \ldots \cup T_m)$. Suppose that V is eliminated on the C_1-side while eliminating W^1; then

$$coef_V(dsum_0^1) = coef_V(\dot{\sum}_{j=1}^m a_j dsum_j^1) \tag{5.13}$$

and V occurs only in $\mathcal{N}_0^1 \cup \ldots \cup \mathcal{N}_m^1$ on the C_1-side. Let $Vsum_j^2$ be the sum of all subterms (each multiplied by some factor) in the constraints of C_2 that involve V and that are in $leaves(\mathbb{T}_{(c_j;B_j)})$ for j in $0..m$ ($Vsum_j^2$ is similar to $dsum_j^1$ that contributes to V on the C_1-side). Then

$$coef_V(c_j) = coef_V(dsum_j^1 \dot{+} Vsum_j^2). \tag{5.14}$$

Since V does not occur in any of the c_j, we have that $coef_V(c_j) = 0$. Combining this with (5.13) and (5.14) yields $coef_V(Vsum_0^2) = coef_V(\dot{\sum}_{j=1}^m a_j Vsum_j^2)$ and hence V can also be eliminated on the C_2-side. □

Using the above lemmas, Proposition 5.3.9 shows that each set label B_0 in a labelled derivation tree over $C_1 \wedge C_2$ can be written as $B_0 = S^1 \cup S^2$ with $S^1 \cup S \in \alpha(C_1)$ and $S^2 \cup S \in \alpha(C_2)$. Afterwards, it is shown in Proposition 5.3.10 that each indirect dependency $D \in \alpha(C_1 \wedge C_2)$ can also be written in that way.

Proposition 5.3.9 (Derivation of direct dependencies)
Let $C_1 \wedge C_2 \in SCons$. Let $(c_0; B_0)$ be a node in a labelled derivation tree $\mathbb{T}$ over $C_1 \wedge C_2$. Then $\exists (S^1 \cup S) \in \alpha(C_1)$, $\exists (S^2 \cup S) \in \alpha(C_2)$ such that $B_0 = S^1 \cup S^2$.

Assume that $leaves(\mathbb{T}_{(c_0;B_0)}) = \{(c_1^1, B_1^1), \ldots, (c_m^1, B_m^1), (c_1^2, B_1^2), \ldots, (c_n^2, B_n^2)\}$ with $B_1^1, \ldots, B_m^1 \in \alpha(C_1)$ and $B_1^2, \ldots, B_n^2 \in \alpha(C_2)$. Then $S^1 \cup S \subseteq B_1^1 \cup \ldots \cup B_m^1$ and $S^2 \cup S \subseteq B_1^2 \cup \ldots \cup B_n^2$. The set S contains (a subset of) the variables that are eliminated during the derivation of B_0. Note that $S^1 \cap S = \emptyset$ and $S^2 \cap S = \emptyset$ since eliminated variables cannot be reintroduced in subsequent derivation steps; S^1 and S^2 are not necessarily disjoint.

PROOF
The proof is trivial if all leaves in $\mathbb{T}_{(c_0;B_0)}$ are related to C_1 and therefore $c_0 \in sform(C_1)$ and $B_0 \in \alpha(C_1)$. Then $S^1 = B_0$ and $S^2 = S = \emptyset$. A similar reasoning holds if all leaves in $\mathbb{T}_{(c_0;B_0)}$ are related to C_2.

Now assume that $\mathbb{T}_{(c_0;B_0)}$ involves leaves of both C_1 and C_2. In that case, the proof is by induction on the depth of $\mathbb{T}_{(c_0;B_0)}$.

Base case : $\mathbb{T}_{(c_0;B_0)}$ is of depth 1. Three cases are distinguished, corresponding to the derivation rules *lu*1, *lu*2 and *ln*1 (in each case below, C_1 and C_2 can be interchanged).

1. Consider $(Y = T_3[Z]\,;\ B_0 = \{Y, Z\})$ that is obtained via *lu*1, i.e. by combining $(X = T_1[Y] \in C_1\,;\ \{X, Y\} \in \alpha(C_1))$ and $(X = T_2[Z] \in C_2\,;\ \{X, Z\} \in \alpha(C_2))$.

Then $S^1 = \{Y\}$, $S^2 = \{Z\}$, $S = \{X\}$ such that $S^1 \cup S \in \alpha(C_1)$, $S^2 \cup S \in \alpha(C_2)$ and $B_0 = S^1 \cup S^2 = \{Y, Z\}$.
Consider $(Y = T_3^+[Z]\ ;\ B_0 = \{Y\})$ that is obtained via *lu*1, i.e. by combining $(X = T_1[Y] \in C_1\ ;\ \{X, Y\} \in \alpha(C_1))$ and $(X = T_2[Z] \in C_2\ ;\ \{X\} \in \alpha(C_2))$.
Then $S^1 = \{Y\}$, $S^2 = \emptyset$, $S = \{X\}$ such that $S^1 \cup S \in \alpha(C_1)$, $S^2 \cup S \in \alpha(C_2)$ and $B_0 = S^1 \cup S^2 = \{Y\}$.

2. Consider $(Y = T_3^+[Z]\ ;\ B_0 = \{Y, Z\})$ that is obtained via *lu*2, i.e. by combining $(X = T_1[Z] \in C_1\ ;\ \{X, Z\} \in \alpha(C_1))$ and $(Y = T_2^+[X] \in C_2\ ;\ \{Y, X\} \in \alpha(C_2))$.
Then $S^1 = \{Z\}$, $S^2 = \{Y\}$, $S = \{X\}$ such that $S^1 \cup S \in \alpha(C_1)$, $S^2 \cup S \in \alpha(C_2)$ and $B_0 = S^1 \cup S^2 = \{Y, Z\}$.

3. Consider $(d_1 = d_2\ ;\ B_0 = vars(d_1, d_2))$ that is obtained via *ln*1, i.e. by combining $(m * e = d' \in C_1\ ;\ vars(e, d') \in \alpha(C_1))$ and $(n * e = d'' \in C_2\ ;\ vars(e, d'') \in \alpha(C_2))$ with $d_1 \doteq (-a*n)*d_1' + (a*m)*d_2'$, $d_2 \doteq (a*n)*d_1'' + (-a*m)*d_2''$ and $vars(d', d'') = vars(d_1, d_2)$.
Then $S^1 = vars(d')$, $S^2 = vars(d'')$, $S = vars(e)$ such that $S^1 \cup S \in \alpha(C_1)$, $S^2 \cup S \in \alpha(C_2)$ and $B_0 = S^1 \cup S^2 = vars(d', d'') = vars(d_1, d_2)$.

Induction : $\mathbb{T}_{(c_0;B_0)}$ is of depth $k > 1$. The induction hypothesis states that for every $(c; B)$ that is the root of a derivation tree of depth $\leq k-1$ it holds that $\exists (T^1 \cup T) \in \alpha(C_1)$, $\exists (T^2 \cup T) \in \alpha(C_2)$ such that $B = T^1 \cup T^2$. Again, three cases are distinguished.

1. Consider $(Y = T_3[Z]\ ;\ B_0 = \{Y, Z\})$ that is obtained via *lu*1, i.e. by combining
   - $(X = T_1[Y] \in C_1\ ;\ \{X, Y\} \in \alpha(C_1))$
     By induction hypothesis:
     $\exists T_1^1 \cup T \in \alpha(C_1)$, $\exists T_1^2 \cup T \in \alpha(C_2) : T_1^1 \cup T_1^2 = \{X, Y\}$.
   - $(X = T_2[Z] \in C_2\ ;\ \{X, Z\} \in \alpha(C_2))$
     By induction hypothesis:
     $\exists T_2^1 \cup T' \in \alpha(C_1)$, $\exists T_2^2 \cup T' \in \alpha(C_2) : T_2^1 \cup T_2^2 = \{X, Z\}$.

   Two situations can occur :

   - $X \in T_1^1 \cup T \cup T_2^1 \cup T'$ and $X \in T_1^2 \cup T \cup T_2^2 \cup T'$ (since $X \notin T \cup T'$ we also have that $X \in T_1^1 \cup T_2^1$ and $X \in T_1^2 \cup T_2^2$), i.e. X occurs both in C_1-constraints and C_2-constraints of $leaves(\mathbb{T}(c_0; B_0))$.
     Since $T_1^1 \cup T, T_2^1 \cup T' \in \alpha(C_1)$, also $T_1^1 \cup T \cup T_2^1 \cup T' \in \alpha(C_1)$ (due to closedness of $\alpha(C_1)$); similarly, $T_1^2 \cup T \cup T_2^2 \cup T' \in \alpha(C_2)$ (due to closedness of $\alpha(C_2)$). Then $S^1 = (T_1^1 \cup T_2^1) \setminus \{X\}$, $S^2 = (T_1^2 \cup T_2^2) \setminus \{X\}$, $S = \{X\} \cup T \cup T'$ such that $S^1 \cup S = T_1^1 \cup T \cup T_2^1 \cup T' \in \alpha(C_1)$ and $S^2 \cup S = T_1^2 \cup T \cup T_2^2 \cup T' \in \alpha(C_2)$ (note: $X \notin T \cup T'$ since an eliminated variable cannot be reintroduced).

   - $X \in T_1^1 \cup T_2^1$ and $X \notin T_1^2 \cup T_2^2$ (so $X \in T_1^1 \cap T_2^2$), i.e. X occurs only in C_1-constraints of $leaves(\mathbb{T}(c_0; B_0))$ (a similar reasoning holds if $X \in T_1^2 \cup T_2^2$ and $X \notin T_1^1 \cup T_2^1$).
     a. X is non-numerical
        Then via Lemma 5.3.3 we have $(T_1^1 \cup T \cup T_2^1 \cup T') \setminus \{X\} \in \alpha(C_1)$.
        By closedness of $\alpha(C_2)$, $T_1^2 \cup T \cup T_2^2 \cup T' \in \alpha(C_2)$.
        Then $S^1 = (T_1^1 \cup T_2^1) \setminus \{X\}$, $S^2 = T_1^2 \cup T_2^2$ and $S = T \cup T'$ such that $S^1 \cup S \in \alpha(C_1)$ (note : $X \notin T \cup T'$ since an eliminated variable cannot be reintroduced;

so $S^1 \cup S = ((T_1^1 \cup T_2^1) \setminus \{X\}) \cup (T \cup T') = (T_1^1 \cup T \cup T_2^1 \cup T') \setminus \{X\})$, $S^2 \cup S \in \alpha(C_2)$ and $B_0 = S^1 \cup S^2 = \{Y, Z\}$.

b. X is numerical

B_0 can be divided into $B_0^1 = (T_1^1 \cup T_2^1) \setminus \{X\}$, containing the variables of B_0 that appear in C_1-constraints in $leaves(\mathbb{T}_{(c_0;B_0)})$, and $B_0^2 = T_1^2 \cup T_2^2$, containing the variables of B_0 that appear in C_2-constraints in $leaves(\mathbb{T}_{(c_0;B_0)})$. Via Lemma 5.3.4,
$(T_1^1 \cup T_2^1 \cup T \cup T') \setminus (\{X\} \cup R^1 \cup R) = (B_0^1 \setminus R^1) \uplus ((T \cup T') \setminus R) \in \alpha(C_1)$,
$(T_1^2 \cup T_2^2 \cup T \cup T') \setminus (R^2 \cup R) = (B_0^2 \setminus R^2) \uplus ((T \cup T') \setminus R) \in \alpha(C_2)$
with $(B_0^1 \setminus R^1) \cup (B_0^2 \setminus R^2) = B_0$.

Recall that R, R^1 and R^2 are the sets of variables that are eliminated while eliminating X; the correspondence between the sets in the lemma and the sets here is :
$vars(e) = \{X\}$;
$E^1 = \{X\}$, $W^1 = \{X\}$, $E^2 = \emptyset$, $W^2 = \emptyset$;
$B_0^1 \setminus R^1 = ((T_1^1 \cup T_2^1) \setminus \{X\}) \setminus R^1 = (T_1^1 \cup T_2^1) \setminus (\{X\} \cup R^1)$;
$B_0^2 \setminus R^2 = (T_1^2 \cup T_2^2) \setminus R^2$.
Then $S^1 = (T_1^1 \cup T_2^1) \setminus (\{X\} \cup R^1)$, $S^2 = (T_1^2 \cup T_2^2) \setminus R^2$ and $S = (T \cup T') \setminus R$ such that $S^1 \cup S \in \alpha(C_1)$, $S^2 \cup S \in \alpha(C_2)$ and $B_0 = S^1 \cup S^2 = \{Y, Z\}$.

A variant of rule *lu*1 is used to derive non-freeness : $(Y = T_3^+[Z]\ ;\ B_0 = \{Y\})$. Then S^1, S^2 and S are constructed in a similar way as above, only now $S^1 \cup S^2 = \{Y\}$.

2. Consider $(Y = T_3^+[Z]; B_0 = \{Y, Z\})$ that is obtained via *lu*2, i.e. by combining

- $(X = T_1[Z]; \{X, Z\})$
  By induction hypothesis:
  $\exists T_1^1 \cup T \in \alpha(C_1)$, $\exists T_1^2 \cup T \in \alpha(C_2)$: $T_1^1 \cup T_1^2 = \{X, Z\}$.
- $(Y = T_2^+[X]; \{Y, X\})$
  By induction hypothesis:
  $\exists T_2^1 \cup T' \in \alpha(C_1)$, $\exists T_2^2 \cup T' \in \alpha(C_2)$: $T_2^1 \cup T_2^2 = \{Y, X\}$.

The construction of S^1, S^2 and S is the same as in case 1 above.

3. Consider $(d_1 = d_2; B_0 = vars(d_1, d_2))$ that is obtained via *ln*1, i.e. by combining

- $(m * e = d'\ ;\ vars(e, d'))$
  By induction hypothesis:
  $\exists T_1^1 \cup T \in \alpha(C_1)$, $\exists T_1^2 \cup T \in \alpha(C_2)$: $T_1^1 \cup T_1^2 = vars(e, d')$.
- $(n * e = d''\ ;\ vars(e, d''))$
  By induction hypothesis:
  $\exists T_2^1 \cup T' \in \alpha(C_1)$, $\exists T_2^2 \cup T' \in \alpha(C_2)$: $T_2^1 \cup T_2^2 = vars(e, d'')$.

Recall that $vars(d', d'') = vars(d_1, d_2)$.

Two situations can occur :

- Each variable of $vars(e)$ occurs both in C_1-constraints and C_2-constraints of $leaves(\mathbb{T}_{(c_0;B_0)})$, i.e. $vars(e) \subseteq T_1^1 \cup T \cup T_2^1 \cup T'$ and $vars(e) \subseteq T_1^2 \cup T \cup T_2^2 \cup T'$.
  Then $S^1 = (T_1^1 \cup T_2^1) \setminus vars(e)$, $S^2 = (T_1^2 \cup T_2^2) \setminus vars(e)$, $S = vars(e) \cup T \cup T'$ such that $S^1 \cup S = T_1^1 \cup T \cup T_2^1 \cup T' \in \alpha(C_1)$ (by closedness of $\alpha(C_1)$), $S^2 \cup S = T_1^2 \cup T \cup T_2^2 \cup T' \in \alpha(C_2)$ (by closedness of $\alpha(C_2)$) and $S^1 \cup S^2 = B_0$.
- B_0 can be divided into $B_0^1 \cup B_0^2$ with $B_0^1 = (T_1^1 \cup T_2^1) \cap B_0$ and $B_0^2 = (T_1^2 \cup T_2^2) \cap B_0$ (so B_0^1 and B_0^2 contain the variables of B_0 that occur resp. in C_1-constraints and

C_2-constraints of $leaves(\mathbb{T}_{(c_0;B_0)})$). Similarly, $vars(e)$ can be divided into $E^1 = (T_1^1 \cup T_2^1) \cap vars(e)$ and $E^2 = (T_1^2 \cup T_2^2) \cap vars(e)$. A subset W^1 of E^1 occurs *only* in C_1-constraints in $leaves(\mathbb{T}_{(c_0;B_0)})$, i.e. $W^1 \subseteq T_1^1$ and $W^1 \subseteq T_2^1$ and $W^1 \cap (T_1^2 \cup T_2^2) = \emptyset$; a subset W^2 of E^2 occurs only in C_2-constraints in $leaves(\mathbb{T}_{(c_0;B_0)})$, i.e. $W^2 \subseteq T_1^2$ and $W^2 \subseteq T_2^2$ and $W^2 \cap (T_1^1 \cup T_2^1) = \emptyset$ (note : either W^1 or W^2 may be empty). So, W^1 must be eliminated on C_1-side and W^2 must be eliminated on C_2-side. By Lemma 5.3.4,

$$(T_1^1 \cup T_2^1 \cup T \cup T') \setminus (W^1 \cup R^1 \cup R) = (B_0^1 \setminus R^1) \,\dot{\cup}\, [((E^1 \setminus W^1) \,\dot{\cup}\, (T \cup T')) \setminus R] \in \alpha(C_1),$$

$$(T_1^2 \cup T_2^2 \cup T \cup T') \setminus (W^2 \cup R^2 \cup R) = (B_0^2 \setminus R^2) \,\dot{\cup}\, [((E^2 \setminus W^2) \,\dot{\cup}\, (T \cup T')) \setminus R] \in \alpha(C_2)$$

with $(B_0^1 \setminus R^1) \cup (B_0^2 \setminus R^2) = B_0$ and $E^1 \setminus W^1 = E^2 \setminus W^2$ (recall that R^1, R^2 and R are the sets of variables that are eliminated due to the elimination of W^1 and W^2). Then $S^1 = B_0^1 \setminus R^1$, $S^2 = B_0^2 \setminus R^2$ and $S = ((E^1 \setminus W^1) \,\dot{\cup}\, (T \cup T')) \setminus R = ((E^2 \setminus W^2) \,\dot{\cup}\, (T \cup T')) \setminus R$ such that $S^1 \cup S \in \alpha(C_1)$, $S^2 \cup S \in \alpha(C_2)$ and $B_0 = S^1 \cup S^2$. □

Proposition 5.3.10 (Derivation of indirect dependencies)
Let $C_1 \wedge C_2 \in SCons$. Let D be an indirect dependency obtained from the root information in two or more labelled derivation trees over $C_1 \wedge C_2$.
Then $\exists (S^1 \cup S) \in \alpha(C_1)$, $\exists (S^2 \cup S) \in \alpha(C_2)$ such that $D = S^1 \cup S^2$.

Assume that $\{(c_1^1, B_1^1), \ldots, (c_m^1, B_m^1), (c_1^2, B_1^2), \ldots, (c_n^2, B_n^2)\}$ is the set of all leaves in the labelled derivation trees involved in establishing D, where $B_1^1, \ldots, B_m^1 \in \alpha(C_1)$ and $B_1^2, \ldots, B_n^2 \in \alpha(C_2)$. Then $S^1 \cup S \subseteq B_1^1 \cup \ldots \cup B_m^1$ and $S^2 \cup S \subseteq B_1^2 \cup \ldots \cup B_n^2$.
The set S contains a subset of the variables that are eliminated during the derivation of D. Note that $S^1 \cap S = \emptyset$ and $S^2 \cap S = \emptyset$ since eliminated variables cannot be reintroduced in subsequent derivation steps.

PROOF
The proof is trivial if all leaf constraints involved in the derivation of D belong to C_1 and therefore $D \in \alpha(C_1)$. Then $S^1 = D$ and $S^2 = S = \emptyset$. A similar reasoning holds if all leaf constraints belong to C_2.
Now assume that the derivation of D involves constraints of both C_1 and C_2. Two cases are distinguished, corresponding to the rules *li*1 and *li*2.

1. Consider $D = \{X, Y\}$ that is obtained via
   - $(X = T_1[Z]; \{X, Z\})$
     By Proposition 5.3.9:
     $\exists T_1^1 \cup T \in \alpha(C_1)$, $\exists T_1^2 \cup T \in \alpha(C_2)$: $T_1^1 \cup T_1^2 = \{X, Z\}$.
   - $(Y = T_2[Z]; \{Y, Z\})$
     By Proposition 5.3.9:
     $\exists T_2^1 \cup T' \in \alpha(C_1)$, $\exists T_2^2 \cup T' \in \alpha(C_2)$: $T_2^1 \cup T_2^2 = \{Y, Z\}$.

   The construction of S^1, S^2 and S is similar as in Proposition 5.3.9, induction, case 1.

- If Z occurs in $T_1^1 \cup T \cup T_2^1 \cup T'$ as well as in $T_1^2 \cup T \cup T_2^2 \cup T'$, then $S^1 = (T_1^1 \cup T_2^1) \setminus \{Z\}$, $S^2 = (T_1^2 \cup T_2^2) \setminus \{Z\}$, $S = \{Z\} \cup T \cup T'$ such that $S^1 \cup S = T_1^1 \cup T \cup T_2^1 \cup T' \in \alpha(C_1)$, $S^2 \cup S = T_1^2 \cup T \cup T_2^2 \cup T' \in \alpha(C_2)$ and $S^1 \cup S^2 = D$.
- If Z occurs only in e.g. T_1^1 and T_2^1 and not in $T_1^2 \cup T_2^2$, then Z must be eliminated on C_1-side; applying Lemma 5.3.3 yields $S^1 = (T_1^1 \cup T_2^1) \setminus \{Z\}$, $S^2 = T_1^2 \cup T_2^2$ and $S = T \cup T'$ with $S^1 \cup S \in \alpha(C_1)$ and $B_0 = S^1 \cup S^2 = \{X, Y\}$, in case Z is non-numerical; if Z is numerical, then $S^1 = (T_1^1 \cup T_2^1) \setminus (\{Z\} \cup R^1)$, $S^2 = (T_1^2 \cup T_2^2) \setminus R^2$ and $S = (T \cup T') \setminus R$ such that $S^1 \cup S \in \alpha(C_1)$, $S^2 \cup S \in \alpha(C_2)$ and $B_0 = S^1 \cup S^2 = \{X, Y\}$ (via Lemma 5.3.4).

2. Consider $D = \{X_1, \ldots, X_m\} \cup vars(d)$ that is obtained via

- $(X_i = T_i^+[Y_i]; \{X_i, Y_i\})$ $(1 \leq i \leq m)$
  By Proposition 5.3.9:
  $\exists T_i^1 \cup T_i \in \alpha(C_1)$, $\exists T_i^2 \cup T_i \in \alpha(C_2)$: $T_i^1 \cup T_i^2 = \{X_i, Y_i\}$.
- $(a_1 Y_1 + \ldots + a_m Y_m = d; \{Y_1, \ldots, Y_m\} \cup vars(d))$
  By Proposition 5.3.9:
  $\exists T_0^1 \cup T_0 \in \alpha(C_1)$, $\exists T_0^2 \cup T_0 \in \alpha(C_2)$: $T_0^1 \cup T_0^2 = \{Y_1, \ldots, Y_m\} \cup vars(d)$.

Two situations can occur :

- Each variable of $\{Y_1, \ldots, Y_m\}$ occurs both in C_1-constraints and C_2-constraints of $leaves(\mathbb{T}(c_0; B_0))$, i.e. $\{Y_1, \ldots, Y_m\} \subseteq T_0^1 \cup T_0 \cup \ldots \cup T_m^1 \cup T_m$ and $\{Y_1, \ldots, Y_m\} \subseteq T_0^2 \cup T_0 \cup \ldots \cup T_m^2 \cup T_m$.
  Then $S^1 = (T_0^1 \cup \ldots \cup T_m^1) \setminus \{Y_1, \ldots, Y_m\}$, $S^2 = (T_0^2 \cup \ldots \cup T_m^2) \setminus \{Y_1, \ldots, Y_m\}$, $S = \{Y_1, \ldots, Y_m\} \cup T_0 \cup \ldots \cup T_m$ such that $S^1 \cup S = T_0^1 \cup T_0 \cup \ldots \cup T_m^1 \cup T_m \in \alpha(C_1)$ (by closedness of $\alpha(C_1)$), $S^2 \cup S = T_0^2 \cup T_0 \cup \ldots \cup T_m^2 \cup T_m \in \alpha(C_2)$ (by closedness of $\alpha(C_2)$) and $S^1 \cup S^2 = D$.

- D can be divided into $D^1 \cup D^2$ with $D^1 = (T_0^1 \cup \ldots \cup T_m^1) \cap D$ and $D^2 = (T_0^2 \cup \ldots \cup T_m^2) \cap D$. Similarly, $\{Y_1, \ldots, Y_m\}$ can be divided into $E^1 = (T_0^1 \cup \ldots \cup T_m^1) \cap \{Y_1, \ldots, Y_m\}$ and $E^2 = (T_0^2 \cup \ldots \cup T_m^2) \cap \{Y_1, \ldots, Y_m\}$. A subset W^1 of E^1 occurs *only* in C_1-constraints involved in the derivation of D (so $W^1 \not\subseteq T_0^2 \cup T_0 \cup \ldots \cup T_m^2 \cup T_m$) and a subset W^2 of E^2 occurs only in C_2-constraints involved in the derivation of D (so $W^2 \not\subseteq T_0^1 \cup T_0 \cup \ldots \cup T_m^1 \cup T_m$). So, W^1 must be eliminated on C_1-side and W^2 must be eliminated on C_2-side. By Lemma 5.3.5,
  $(T_0^1 \cup T_0 \cup \ldots \cup T_m^1 \cup T_m) \setminus (W^1 \cup R) =$
  $\quad D^1 \,\mathfrak{G}\, [((E^1 \setminus W^1) \,\mathfrak{G}\, (T_0 \cup \ldots \cup T_m)) \setminus R] \in \alpha(C_1)$,
  $(T_0^2 \cup T_0 \cup \ldots \cup T_m^2 \cup T_m) \setminus (W^2 \cup R) =$
  $\quad D^2 \,\mathfrak{G}\, [((E^2 \setminus W^2) \,\mathfrak{G}\, (T_0 \cup \ldots \cup T_m)) \setminus R] \in \alpha(C_2)$
  with $E^1 \setminus W^1 = E^2 \setminus W^2$ (recall that R is the set of variables that are eliminated due to the elimination of W^1 and W^2).
  Then $S^1 = D^1$, $S^2 = D^2$ and $S = ((E^1 \setminus W^1) \,\mathfrak{G}\, (T_0 \cup \ldots \cup T_m)) \setminus R = ((E^2 \setminus W^2) \,\mathfrak{G}\, (T_0 \cup \ldots \cup T_m)) \setminus R$ such that $S^1 \cup S \in \alpha(C_1)$, $S^2 \cup S \in \alpha(C_2)$ and $D = S^1 \cup S^2$. □

Theorem 5.3.1
Let $C_1, C_2 \in Cons$. Then $\alpha(C_1 \wedge C_2) \leq^{\mathcal{F}} \alpha(C_1) \wedge \alpha(C_2)$.

PROOF
The proof is trivial if $C_1 \wedge C_2$ is unsatisfiable. Then $\alpha(C_1 \wedge C_2) = \bot$ and $\bot$ is the minimal element in $Con^{\mathcal{F}}$.
Now assume that $C_1 \wedge C_2$ is satisfiable. Then $\leq^{\mathcal{F}}$ means $\subseteq$.
For each direct dependency B_0 in $\alpha(C_1 \wedge C_2)$, there exists a labelled derivation tree $\mathbb{T}_{(c_0;B_0)}$ over $C_1 \wedge C_2$ with $c_0 \in sform(C_1 \wedge C_2)$. By Proposition 5.3.9 we know that $\exists(S^1 \cup S) \in \alpha(C_1)$, $\exists(S^2 \cup S) \in \alpha(C_2)$ such that $B_0 = S^1 \cup S^2$. This implies that $B_0 \in \alpha(C_1) \wedge \alpha(C_2)$ by definition of abstract conjunction (note : if $S^2 = S = \emptyset$, then $B_0 \in \alpha(C_1)$; if $S^1 = S = \emptyset$, then $B_0 \in \alpha(C_2)$; otherwise, $B_0 \in \alpha(C_1) \oplus \alpha(C_2)$).
Similarly, for each indirect dependency D in $\alpha(C_1 \wedge C_2)$, we have shown in Proposition 5.3.10 that $\exists(S^1 \cup S) \in \alpha(C_1)$, $\exists(S^2 \cup S) \in \alpha(C_2)$ such that $D = S^1 \cup S^2$. This implies that $D \in \alpha(C_1) \wedge \alpha(C_2)$ by definition of abstract conjunction (note : similar to the one above).
□

Corollary 5.3.1
Let $C_1, C_2 \in Cons$ and $AC_1, AC_2 \in Con^{\mathcal{F}}$. If $\alpha(C_1) \leq^{\mathcal{F}} AC_1$ and $\alpha(C_2) \leq^{\mathcal{F}} AC_2$, then $\alpha(C_1 \wedge C_2) \leq^{\mathcal{F}} AC_1 \wedge AC_2$.

PROOF

$\alpha(C_1 \wedge C_2)$	$\leq^{\mathcal{F}}$	$\alpha(C_1) \wedge \alpha(C_2)$	by Theorem 5.3.1
	$\leq^{\mathcal{F}}$	$AC_1 \wedge AC_2$	since $\alpha(C_1) \leq^{\mathcal{F}} AC_1$ and $\alpha(C_2) \leq^{\mathcal{F}} AC_2$ and $\wedge$ preserves order (Proposition 5.3.8).

□

Theorem 5.3.1 and Corollary 5.3.1 can easily be lifted to sets of constraints :

Theorem 5.3.2
Let $CS_1, CS_2 \in Con^{\mathcal{C}}$. Then $\alpha(CS_1 \wedge CS_2) \leq^{\mathcal{F}} \alpha(CS_1) \wedge \alpha(CS_2)$.

Corollary 5.3.2 (Safety of abstract conjunction)
Let $CS_1, CS_2 \in Con^{\mathcal{C}}$ and $AC_1, AC_2 \in Con^{\mathcal{F}}$. If $\alpha(CS_1) \leq^{\mathcal{F}} AC_1$ and $\alpha(CS_2) \leq^{\mathcal{F}} AC_2$, then $\alpha(CS_1 \wedge CS_2) \leq^{\mathcal{F}} (AC_1 \wedge AC_2)$.

5.3.4 Properties of projection

In this section, we prove the safety of abstract projection and point out some other crucial properties of the concrete and abstract projection.

Proposition 5.3.11
Let $CS \in Con^{\mathcal{C}}$ and $V \subseteq Var$. Then $\alpha(\overline{\exists}_V CS) = \overline{\exists}_V \alpha(CS)$.

PROOF
If $CS = \emptyset$, then $\overline{\exists}_V CS = \emptyset$. So $\alpha(\overline{\exists}_V CS) = \overline{\exists}_V \alpha(CS) = \bot$.

Now assume that $CS \neq \emptyset$.

$\alpha(\overline{\exists}_V CS) = \alpha(\{\overline{\exists}_V C \mid C \in CS\})$ by definition of concrete projection
$= \bigcup_{C \in CS} \alpha(\{\overline{\exists}_V C\})$ by Definition 5.2.9 of the abstraction function
$= \bigcup_{C \in CS} \{S \in \alpha(C) \mid S \subseteq V\}$ by Definition 5.2.14 (abstraction of a projected constraint)
$= \{S \in \alpha(CS) \mid S \subseteq V\}$ by Definition 5.2.9 of the abstraction function
$= \overline{\exists}_V \alpha(CS)$ by Definition 5.3.2 of abstract projection

□

Proposition 5.3.12 (Abstract projection preserves order)
Let $AC_1, AC_2 \in Con^{\mathcal{F}}$. If $AC_1 \leq^{\mathcal{F}} AC_2$, then $\overline{\exists}_V AC_1 \leq^{\mathcal{F}} \overline{\exists}_V AC_2$.

PROOF
If $AC_1 = \bot$ or ($AC_2 = \bot$ and therefore also $AC_1 = \bot$), then the proof is trivial. Otherwise, $AC_1 \leq^{\mathcal{F}} AC_2$ means that $AC_1 \subseteq AC_2$. Then

$\overline{\exists}_V AC_1 = \{S \in AC_1 \mid S \subseteq V\}$ (by Definition 5.3.2 of abstract projection)
$\subseteq \{S \in AC_2 \mid S \subseteq V\}$ (since $AC_1 \subseteq AC_2$)
$= \overline{\exists}_V AC_2$ (by Definition 5.3.2 of abstract projection)

□

Theorem 5.3.3 (Safety of abstract projection)
Let $CS \in Con^{\mathcal{C}}$ and $AC \in Con^{\mathcal{F}}$. If $\alpha(CS) \leq^{\mathcal{F}} AC$, then $\alpha(\overline{\exists}_V CS) \leq^{\mathcal{F}} \overline{\exists}_V AC$.

PROOF
If $CS = \emptyset$, then $\overline{\exists}_V CS = \emptyset$ and $\alpha(\overline{\exists}_V CS) = \bot$. Since $\bot$ is the minimal element in $Con^{\mathcal{F}}$, $\alpha(\overline{\exists}_V CS) \leq^{\mathcal{F}} \overline{\exists}_V AC$.
Now assume that $CS \neq \emptyset$. Then $\alpha(CS) \leq^{\mathcal{F}} AC$ means that $\alpha(CS) \subseteq AC$.

$\alpha(\overline{\exists}_V CS) = \overline{\exists}_V \alpha(CS)$ by Proposition 5.3.11
$\subseteq \overline{\exists}_V AC$ by $\alpha(CS) \subseteq AC$ and Proposition 5.3.12.

□

The definition of concrete projection implies the following properties.

Property 5.3.1
Let $C, C' \in SCons$ and $V \subseteq Var$.

1. *$C \Rightarrow \overline{\exists}_V C$.*
2. *If $vars(C) \subseteq V$, then $\overline{\exists}_V C = C$.*
3. *If $vars(C') \subseteq V$, then $\overline{\exists}_V (C \wedge C') = \overline{\exists}_V C \wedge C'$.*
4. *If $C_1 \Rightarrow C_2$, then $\overline{\exists}_V C_1 \Rightarrow \overline{\exists}_V C_2$.*

For more details, we refer to [46] where these properties are shown to hold for CLP(R) (generalisation to CLP(H,N) is straightforward).

Propositions 5.3.13 and 5.3.14 below express that projection does not cause loss of information. First of all, concrete projection is considered. If the computation is in a state where only constraints over V are to be added, then no pruning will be lost when the store is projected onto the variables of V and the computation is continued with the reduced store. The existence of projections with these properties has to do with the inherent algebraic properties of the constraint domain (cf. [46] for more details).

Proposition 5.3.13
Let $C \in SCons$ and $V \subseteq Var$. Then,

1. $\forall\ C' \in Con^{\mathcal{C}}$ over V (i.e. $vars(C') \subseteq V$): $C \Rightarrow C'$ iff $\exists_V C \Rightarrow C'$.
2. $\forall\ C', C'' \in Con^{\mathcal{C}}$ over V : $(C \wedge C') \Rightarrow C''$ iff $(\exists_V C \wedge C') \Rightarrow C''$.

PROOF

1. Applying Property 5.3.1 (point 4), we obtain : if $C \Rightarrow C'$, then $\exists_V C \Rightarrow \exists_V C'$. Herein, $\exists_V C' = C'$ since $vars(C') \subseteq V$ (Property 5.3.1, point 2). So, if $C \Rightarrow C'$, then $\exists_V C \Rightarrow C'$. Vice versa, if $\exists_V C \Rightarrow C'$, then $C \Rightarrow C'$ since $C \Rightarrow \exists_V C$ (Property 5.3.1, point 1) and " $\Rightarrow$ " is transitive.

2. Substitute C'' for C' and $C \wedge C'$ for C in case 1. This yields : $\forall C', C'' \in Con^{\mathcal{C}}$ over V : $(C \wedge C') \Rightarrow C''$ iff $\exists_V (C \wedge C') \Rightarrow C''$. Since C' is a constraint over V, it holds that $\exists_V (C \wedge C') = \exists_V C \wedge C'$ (Property 5.3.1, point 3). Hence, $\forall C', C'' \in Con^{\mathcal{C}}$ over V : $(C \wedge C') \Rightarrow C''$ iff $(\exists_V C \wedge C') \Rightarrow C''$. □

The proposition can be lifted to sets of constraints (i.e. $CS, CS', CS'' \in Con^{\mathcal{C}}$) in a straightforward way.

Abstract projection does not cause loss of information either. In particular, the abstract projection on V cannot invalidate the inferred freeness and dependencies of the variables in V.

Proposition 5.3.14
Let $AC \in Con^{\mathcal{F}}$ and $V \subseteq Var$. Then

1. $\forall\ AC' \in Con^{\mathcal{F}}$ over V (i.e. $\forall S \in AC' : S \subseteq V$): $AC' \leq^{\mathcal{F}} AC$ iff $AC' \leq^{\mathcal{F}} \exists_V AC$;
2. $\forall\ AC', AC'' \in Con^{\mathcal{F}}$ over V : $AC'' \leq^{\mathcal{F}} (AC \wedge AC')$ iff $AC'' \leq^{\mathcal{F}} (\exists_V AC \wedge AC')$.

PROOF

1. If $AC' = \bot$, or if $AC = \bot$ (and by $AC' \leq^{\mathcal{F}} AC$ therefore also $AC' = \bot$), the proof is straightforward.
Now assume that AC and $AC' \neq \bot$; in that case $\leq^{\mathcal{F}}$ means $\subseteq$.
(proof of $\Rightarrow$)
AC' is an abstract constraint over V, so $\forall S \in AC' : S \subseteq V$. Given is also that $AC' \subseteq AC$. So, $\forall S \in AC' : S \subseteq V$ *and* $S \in AC$. By Definition 5.3.2, $\exists_V AC$ is the set of *all* T such that $T \in AC$ *and* $T \subseteq V$. Hence, $AC' \subseteq \exists_V AC$.
(proof of $\Leftarrow$)
$AC' \subseteq \exists_V AC =$ (by Definition 5.3.2 of abstract projection) $\{S \in AC \mid S \subseteq V\} \subseteq AC$.

2. Substitute AC'' for AC' and $AC \wedge AC'$ for AC in case 1. This yields :
$\forall AC', AC'' \in Con^{\mathcal{F}}$ over V : $AC'' \leq^{\mathcal{F}} (AC \wedge AC')$ iff $AC'' \leq^{\mathcal{F}} \exists_V (AC \wedge AC')$. It then has to be proved that $\exists_V (AC \wedge AC') = \exists_V AC \wedge AC'$. The proof is straightforward if AC and/or AC' equal $\bot$. Otherwise, the reasoning is as follows.

$$AC \wedge AC' = AC \cup AC' \cup \left(\{(A_1 \cup A_2) \setminus D \mid A_1 \in AC, A_2 \in AC', D \subseteq (A_1 \cap A_2)\} \setminus \{\emptyset\} \right)$$

(by Definition 5.3.1 of abstract conjunction).

Note that $\overline{\exists}_V(\bigcup_{i=1}^n B_i) = \bigcup_{i=1}^n(\overline{\exists}_V B_i)$ (follows immediately from Definition 5.3.2 of abstract projection). So,

$$\begin{aligned}\overline{\exists}_V(AC \wedge AC') &= \overline{\exists}_V AC \cup \overline{\exists}_V AC' \cup \\ &\quad (\{S \mid S = (A_1 \cup A_2) \setminus D, A_1 \in AC, A_2 \in AC', \\ &\qquad D \subseteq (A_1 \cap A_2), S \subseteq V\} \setminus \{\emptyset\}).\end{aligned}$$

In this formula, $\overline{\exists}_V AC' = AC'$ since AC' is an abstract constraint over V. The information that $(S = (A_1 \cup A_2) \setminus D, A_2 \in AC'(over\ V), D \subseteq (A_1 \cap A_2), S \subseteq V)$ implies that $A_1 \subseteq V$; this, together with the fact that $A_1 \in AC$, implies that $A_1 \in \overline{\exists}_V AC$. The above formula is thus transformed into :

$$\begin{aligned}\overline{\exists}_V(AC \wedge AC') &= \overline{\exists}_V AC \cup AC' \cup \\ &\quad (\{S \mid S = (A_1 \cup A_2) \setminus D, A_1 \in \overline{\exists}_V AC, A_2 \in AC', \\ &\qquad D \subseteq (A_1 \cap A_2)\} \setminus \{\emptyset\}) \\ &= \overline{\exists}_V AC \wedge AC' \\ &\qquad \text{(by Definition 5.3.1 of abstract conjunction).}\end{aligned}$$

□

5.4 Abstract operations

In this section, we define the higher-level abstract operations in the abstract interpretation framework (abstract interpretation of a constraint, procedure-entry, procedure-exit). We formally prove their safety with respect to the concrete execution. The operations make use of the primitive abstract operations defined in Section 5.3.

To simplify the definitions of the operations, we do not explicitly deal with the $\bot$ case. Note that as soon as $\bot$ is obtained as the abstraction of a constraint at some point, $\bot$ is propagated over all the program points up to the end of the clause. If the abstract constraints at the end of all clauses defining a procedure equal $\bot$, then $\bot$ is also propagated to the calling environment.

5.4.1 Compound abstract constraints

Before defining the abstract operations we must point out an issue related to the precision of the analysis.

An abstract constraint is split into two components to distinguish between *old* information that is passed down from a calling environment at procedure-entry, and *new* information that is gathered during local analysis of a clause body. The distinction plays a major role in the precision of the procedure-exit operation. Upon procedure-exit, it is essential not to compose the old information with the information in the caller's environment, as this would destroy the freeness of the involved variables. The reason is that, as our abstraction discards the coefficients of numerical constraints, it is disastrous to inadvertently add a numerical constraint twice to the abstract constraint store. Consider a constraint $a_1X_1 + \ldots + a_nX_n = a_{n+1}$ which is abstracted as $\{\{X_1, \ldots, X_n\}\}$. Adding this constraint a second time (i.e. joining its abstraction with a store AC already containing $\{X_1, \ldots, X_n\}$) results in a store AC' containing $\{X_i\}$ for each of the variables X_i; hence, AC' indicates that each X_i is possibly non-free. The computation of AC' can be explained as follows in terms of

combining equations. The set $\{X_1, \ldots, X_n\}$ in AC represents an equation of the form $b_1X_1 + \ldots + b_nX_n = b_{n+1}$ where the value of the coefficients b_i is no longer known. This equation is then combined with $a_1X_1 + \ldots + a_nX_n = a_{n+1}$. With an adequate choice of the b_i, one can obtain a linear combination that contains only X_i.
Making the distinction between new and old information in the analysis of logic programs has been applied previously by Plaisted [99] and also by Mulkers [91, 92].

The abstract domain now consists of *compound abstract constraints* rather than of abstract constraints as such. We formally define a compound abstract constraint and the basic operations on it. When no confusion is possible, a compound abstract constraint is simply referred to as "abstract constraint" in the sequel.

Definition 5.4.1 (Compound abstract constraint)
A compound abstract constraint AC is either $\bot$ or a pair (AC^o, AC^n) with $AC^o, AC^n \in \wp(\wp_0(Var))$. The component AC^o contains the information passed on at procedure-entry and the combination of it with the information gathered during local analysis of the procedure body; AC^n contains the information that is obtained from the constraints gathered during local analysis of the procedure body.

Note that AC^o and AC^n are not necessarily disjoint.

Definition 5.4.2 (AC^t)
Let (AC^o, AC^n) be a compound abstract constraint. Then $AC^t = AC^o \cup AC^n$.

To infer mode and dependency information from a compound abstract constraint, the distinction between the o(ld) and n(ew) components is not relevant; it is sufficient to consider the complete abstract constraint AC^t. Proposition 5.2.3 formally describes the relation between AC^t and the modes and dependencies of variables.

Definition 5.4.3 (Order)
Let AC_1 and AC_2 be two compound abstract constraints. Then[20],

$$AC_1 \leq^{\mathcal{F}} AC_2 \quad \text{iff} \quad AC_1 = \bot \text{ or } (AC_1 = (AC_1^o, AC_1^n), AC_2 = (AC_2^o, AC_2^n) \text{ and } AC_1^o \subseteq AC_2^o \text{ and } AC_1^n \subseteq AC_2^n).$$

Definition 5.4.4 (Least upper bound)
Let AC_1 and AC_2 be two compound abstract constraints. Then,

$$AC_{lub} = lub^{\mathcal{F}}(AC_1, AC_2) = \begin{cases} AC_2 & \text{if } AC_1 = \bot \\ AC_1 & \text{if } AC_2 = \bot \\ (AC_1^o \cup AC_2^o, AC_1^n \cup AC_2^n) & \text{if } AC_1 = (AC_1^o, AC_1^n) \text{ and } AC_2 = (AC_2^o, AC_2^n) \end{cases}$$

[20]The symbol $\leq^{\mathcal{F}}$ is overloaded. Depending on the context, it denotes the order between compound abstract constraints or between plain abstract constraints (Definition 5.2.6).

When using the abstract interpretation system PLAI of Muthukumar and Hermenegildo [96, 94], the above least upper bound operation is only needed when annotating the program: normally only *one* general version of each predicate is given as output. This implies that for each point in a predicate the system has to compute the upper bound of all abstract constraints generated at that point in the different predicate specialisations. During the analysis phase however (cf. Definition 5.4.7 of procedure-exit), only the AC^n_{lub}-component is needed and is computed using Definition 5.2.7.

The greatest lower bound can be defined accordingly.

5.4.2 Abstract interpretation of a constraint

The abstract interpretation of a constraint C consists of computing the abstraction of C, $\alpha(C)$, and joining it with the current abstract constraint store. Hereby, $\alpha(C)$ itself and the combination of it with the local information is put into the n-component, whereas the combination with information passed on at procedure-entry is put into the o-component.

Definition 5.4.5 (Abstract interpretation of a constraint)
Let $C \in SCons$ and let (AC^o_c, AC^n_c) be the abstract call constraint of C. The abstract success constraint (AC^o_s, AC^n_s) of C is defined as :

1. $AC^o_s = AC^o_c \cup (AC^o_c \oplus \alpha(C))$ *and*
2. $AC^n_s = AC^n_c \wedge \alpha(C)$.

Lemma 5.4.1
Let $C \in SCons$ and let (AC^o_c, AC^n_c) and (AC^o_s, AC^n_s) be resp. the abstract call and success constraint of C. Then, $AC^t_s = AC^t_c \wedge \alpha(C)$; moreover, $\alpha(C) \subseteq AC^n_s$, $AC^n_c \oplus \alpha(C) \subseteq AC^n_s$, $AC^o_c \oplus \alpha(C) \subseteq AC^o_s$, $AC^o_c \subseteq AC^o_s$ and $AC^n_c \subseteq AC^n_s$.

PROOF
$AC^o_s = AC^o_c \cup (AC^o_c \oplus \alpha(C))$ and $AC^n_s = AC^n_c \wedge \alpha(C) = AC^n_c \cup \alpha(C) \cup (AC^n_c \oplus \alpha(C))$ by Definition 5.4.5 (abstract interpretation of a constraint). By distributivity of $\oplus$ over $\cup$ (Proposition 5.3.1), $(AC^o_c \oplus \alpha(C)) \cup (AC^n_c \oplus \alpha(C)) = AC^t_c \oplus \alpha(C)$. So $AC^t_s = AC^o_s \cup AC^n_s = AC^t_c \cup \alpha(C) \cup (AC^t_c \oplus \alpha(C)) = AC^t_c \wedge \alpha(C)$ (by Definition 5.3.1 of abstract conjunction). Moreover, it follows immediately from Definition 5.4.5 and Definition 5.3.1 that $\alpha(C) \subseteq AC^n_s$, $AC^n_c \oplus \alpha(C) \subseteq AC^n_s$, $AC^o_c \oplus \alpha(C) \subseteq AC^o_s$, $AC^o_c \subseteq AC^o_s$ and $AC^n_c \subseteq AC^n_s$. □

Proposition 5.4.1
Let $C \in SCons$ and let (AC^o_c, AC^n_c) and (AC^o_s, AC^n_s) be resp. the abstract call and success constraint of C. For each $C_c \in SCons$ holds : if $\alpha(C_c) \subseteq AC^t_c$, then $\alpha(C_c \wedge C) \subseteq AC^t_s$.

PROOF
By Corollary 5.3.1 (concerning the safety of abstract conjunction), it holds for every $C_c \in SCons$ that $\alpha(C_c \wedge C) \subseteq (AC^t_c \wedge \alpha(C))$ since $\alpha(C_c) \subseteq AC^t_c$ and $\alpha(C) \subseteq \alpha(C)$. Applying Lemma 5.4.1, $AC^t_c \wedge \alpha(C) = AC^t_s$. So, for every $C_c \in SCons$, $\alpha(C_c) \subseteq AC^t_c$ implies $\alpha(C_c \wedge C) \subseteq AC^t_s$. □

The above proposition can easily be lifted to sets of constraints :

Corollary 5.4.1 (Safety of abstract interpretation of a constraint)
Let $CS \in Con^c$, $C \in SCons$ and let (AC_c^o, AC_c^n) and (AC_s^o, AC_s^n) be resp. the abstract call and success constraint of C. If $\alpha(CS) \subseteq AC_c^t$, then $\alpha(CS \wedge \{C\}) \subseteq AC_s^t$.

PROOF
Straightforward □

5.4.3 Procedure-entry

Procedure-entry is the first step in the abstract interpretation of a call $p(Y_1, \ldots, Y_k)$[21]. Assume that the variables in the calling environment are $\{X_1, \ldots, X_m, Y_1, \ldots, Y_k\}$. The purpose of procedure-entry is to initialise the abstract constraint of the clause that is used to resolve the call. To do so, the constraint over $\{X_1, \ldots, X_m, Y_1, \ldots, Y_k\}$ is projected onto $\{Y_1, \ldots, Y_k\}$ and the projected constraint is renamed, which yields the o-component of the abstract constraint at the beginning of the clause. The n-component is initialised to the empty set.

Definition 5.4.6 (Procedure-entry)
Let (AC_c^o, AC_c^n) be the abstract call constraint of a procedure call $p(Y_1, \ldots, Y_k)$ and let $p(Z_1, \ldots, Z_k)$ be the head of a clause[22] used to resolve the call. Then, the abstract call constraint of the clause is defined as $(AC_{in}^o, AC_{in}^n) = (AC_{entry}^o, AC_{entry}^n)\rho$ with $AC_{entry} = (\overline{\exists}_{\{Y_1,\ldots,Y_k\}} AC_c^t, \emptyset)$ and $\rho = \{Y_1 \leftarrow Z_1, \ldots, Y_k \leftarrow Z_k\}$ (renaming).

Recall that AC_{entry} is needed when dealing with recursive calls (Section 3.2.4). However, the safety of procedure-entry will be proven in a single step (since the only difference between AC_{entry} and AC_{in} is the application of ρ).

Proposition 5.4.2 (Safety of procedure-entry)
Let (AC_c^o, AC_c^n) be the abstract call constraint of a procedure call $p(Y_1, \ldots, Y_k)$ and let $p(Z_1, \ldots, Z_k)$ be the head of a clause used to resolve the call. Let $\rho = \{Y_1 \leftarrow Z_1, \ldots, Y_k \leftarrow Z_k\}$ (renaming). Let (AC_{in}^o, AC_{in}^n) be the abstract call constraint at the beginning of the clause. For every $C_c \in Con_c$ holds : if $\alpha(C_c) \subseteq AC_c^t$, then $\alpha((\overline{\exists}_{\{Y_1,\ldots,Y_k\}} C_c)\rho) \subseteq AC_{in}^t$ (in fact, $\subseteq AC_{in}^o$).

PROOF
$AC_{in}^t = AC_{in}^o = (\overline{\exists}_{\{Y_1,\ldots,Y_k\}} AC_c^t)\rho$ by Definition 5.4.6 of procedure-entry.
If $\alpha(C_c) \subseteq AC_c^t$, then $\alpha(\overline{\exists}_{\{Y_1,\ldots,Y_k\}} C_c) \subseteq \overline{\exists}_{\{Y_1,\ldots,Y_k\}} AC_c^t$ by Theorem 5.3.3 (safety of abstract projection). After renaming one obtains : $\alpha((\overline{\exists}_{\{Y_1,\ldots,Y_k\}} C_c)\rho) \subseteq (\overline{\exists}_{\{Y_1,\ldots,Y_k\}} AC_c^t)\rho$. Combining the above yields : if $\alpha(C_c) \subseteq AC_c^t$, then $\alpha((\overline{\exists}_{\{Y_1,\ldots,Y_k\}} C_c)\rho) \subseteq AC_{in}^t$ (more precisely, $\subseteq AC_{in}^o$). □

[21] Programs are assumed to be normalised, i.e. all unifications and numerical constraints are made explicit such that the arguments of the predicates are distinct variables.
[22] The clause is assumed to be renamed apart from $vars(AC_c^t)$.

5.4.4 Procedure-exit

Procedure-exit is the last step in the abstract interpretation of a call $p(Y_1,\ldots,Y_k)$. It is applied when the final abstract constraint has been derived for each clause defining p/k. Let Cl_i be the i^{th} clause defining p/k $(1 \leq i \leq m)$ and let (AC_i^o, AC_i^n) be the abstract constraint at the end of Cl_i. Assume that the head of each Cl_i is of the form $p(Z_1,\ldots,Z_k)$. First of all, each (AC_i^o, AC_i^n) is projected onto the variables $Z_1,\ldots,Z_k$ of the head and is then renamed to the variables $Y_1,\ldots,Y_k$ of the call. Then the least upper bound of the resulting constraints is computed, yielding $(AC_{exit}^o, AC_{exit}^n)$. In fact, only AC_{exit}^n is needed since this component contains the information derived during execution of the call itself (AC_{exit}^o contains the information passed down at procedure-entry and combinations of it with the local information). Finally, the extension step combines the abstract call constraint of $p(Y_1,\ldots,Y_k)$ and AC_{exit}^n to obtain the success constraint of the call. As mentioned in Section 5.4.1, it is important not to combine the information that was passed down at procedure-entry (i.e. information in AC_{exit}^o) once again with the call constraint, as this would have a disastrous effect on precision.

Definition 5.4.7 (Procedure-exit)
Let (AC_c^o, AC_c^n) be the abstract call constraint of a call $p(Y_1,\ldots,Y_k)$. Let the head of each clause Cl_i defining p/k be of the form $p(Z_1,\ldots,Z_k)$ $(1 \leq i \leq m)$. Let (AC_i^o, AC_i^n) be the abstract constraint at the end of Cl_i and $AC_{exit}^n = lub^{\mathcal{F}}(\{\ldots,(\exists_{\{Z_1,\ldots,Z_k\}} AC_i^n)\rho^{-1},\ldots\})$ with $\rho^{-1} = \{Z_1 \leftarrow Y_1,\ldots,Z_k \leftarrow Y_k\}$ (renaming). The abstract success constraint (AC_s^o, AC_s^n) of the call is defined as :

1. $AC_s^o = AC_c^o \cup (AC_c^o \oplus AC_{exit}^n)$
2. $AC_s^n = AC_c^n \wedge AC_{exit}^n$

Note that part of the information computed during analysis of the procedure is recomputed at procedure-exit. At procedure-entry, the parts of AC_c^o and AC_c^n that are related to the call variables are passed down. More precisely, $(\exists_V AC_c^t)\rho$ is put into the o-component of the abstract constraint at the beginning of each clause defining the procedure. During local analysis of the procedure, this information is combined with the information that is gathered locally and that is finally in AC_{exit}^n; the results of the combination are put into the o-component of the abstract constraints and can finally be found in AC_{exit}^o. However, since one cannot separate out $(\exists_V AC_c^o) \oplus AC_{exit}^n$, which should be in AC_s^o, and $(\exists_V AC_c^n) \oplus AC_{exit}^n$, which should be in AC_s^n, this information is recomputed at procedure-exit (as part of $AC_c^o \oplus AC_{exit}^n$ and $AC_c^n \oplus AC_{exit}^n$).

Lemma 5.4.2
Assume the same definitions for (AC_c^o, AC_c^n), (AC_s^o, AC_s^n), (AC_i^o, AC_i^n), $(AC_{exit}^o, AC_{exit}^n)$ and Cl_i as in Definition 5.4.7. Let $V = \{Y_1,\ldots,Y_k\}$, $W = \{Z_1,\ldots,Z_k\}$ and $\rho = \{Y_1 \leftarrow Z_1,\ldots,Y_k \leftarrow Z_k\}$ (renaming). Let $C_{local\_i} \in SCons$ be the constraint added during execution of Cl_i (projected onto $vars(Cl_i)$) and $C'_{local\_i} = (\exists_W C_{local\_i})\rho^{-1}$.
If $(\exists_V AC_c^t)\rho \wedge \alpha(C_{local\_i}) \subseteq AC_i^t$ with
$\alpha(C_{local\_i}) \subseteq AC_i^n$ and $(\exists_V AC_c^t)\rho \oplus \alpha(C_{local\_i}) \subseteq AC_i^o$,
then $AC_c^t \wedge \alpha(C'_{local\_i}) \subseteq AC_s^t$
with $\alpha(C'_{local\_i}) \subseteq AC_s^n$, $AC_c^n \oplus \alpha(C'_{local\_i}) \subseteq AC_s^n$, $AC_c^o \oplus \alpha(C'_{local\_i}) \subseteq AC_s^o$,
$AC_c^o \subseteq AC_s^o$ and $AC_c^n \subseteq AC_s^n$.

PROOF
$AC_c^t \wedge \alpha(C'_{local\_i}) = AC_c^t \cup \alpha(C'_{local\_i}) \cup (AC_c^t \oplus \alpha(C'_{local\_i}))$ by Definition 5.3.1 of abstract conjunction.

1. $\underline{AC_c^t \subseteq AC_s^t}$ (by Definition 5.4.7 of procedure-exit)
2. $\underline{\alpha(C'_{local\_i}) \subseteq AC_{exit}^n \subseteq AC_s^n}$
   Given is $\alpha(C_{local\_i}) \subseteq \mathcal{AC}_i^n$. Applying projection and renaming on both sides yields $(\exists_W \alpha(C_{local\_i}))\rho^{-1} \subseteq (\exists_W \mathcal{AC}_i^n)\rho^{-1}$. Herein, $(\exists_W \alpha(C_{local\_i}))\rho^{-1} = \alpha((\exists_W C_{local\_i})\rho^{-1})$ by Proposition 5.3.11 (related to the safety of abstract projection) and $\alpha((\exists_W C_{local\_i})\rho^{-1}) = \alpha(C'_{local\_i})$; also, $(\exists_W \mathcal{AC}_i^n)\rho^{-1} \subseteq \mathcal{AC}_{exit}^n$ (by Definition 5.4.7 of procedure-exit and Definition 5.2.7 of least upper bound) and $\mathcal{AC}_{exit}^n \subseteq \mathcal{AC}_s^n$ (by Definition 5.4.7).
   Combining the above yields $\alpha(C'_{local\_i}) \subseteq \mathcal{AC}_{exit}^n \subseteq \mathcal{AC}_s^n$.
3. $\underline{(AC_c^t \oplus \alpha(C'_{local\_i})) \subseteq AC_s^t}$
   a. proof of $AC_c^n \oplus \alpha(C'_{local\_i}) \subseteq AC_s^n$
      $AC_c^n \oplus \alpha(C'_{local\_i}) \subseteq AC_c^n \oplus AC_{exit}^n$ since $\alpha(C'_{local\_i}) \subseteq AC_{exit}^n$ (cf. case 2) and $\oplus$ preserves order (Proposition 5.3.6). $AC_c^n \oplus AC_{exit}^n \subseteq AC_s^n$ by Definition 5.4.7 of procedure-exit. So, $AC_c^n \oplus \alpha(C'_{local\_i}) \subseteq AC_s^n$.
   b. proof of $AC_c^o \oplus \alpha(C'_{local\_i}) \subseteq AC_s^o$
      $AC_c^o \oplus \alpha(C'_{local\_i}) \subseteq AC_c^o \oplus AC_{exit}^n$ since $\alpha(C'_{local\_i}) \subseteq AC_{exit}^n$ (cf. case 2) and $\oplus$ preserves order (Proposition 5.3.6). $AC_c^o \oplus AC_{exit}^n \subseteq AC_s^o$ by Definition 5.4.7 of procedure-exit. So, $AC_c^o \oplus \alpha(C'_{local\_i}) \subseteq AC_s^o$.

   Cases a and b imply that $(AC_c^o \oplus \alpha(C'_{local\_i})) \cup (AC_c^n \oplus \alpha(C'_{local\_i})) \subseteq AC_s^t$. By distributivity of $\oplus$ over $\cup$ (Proposition 5.3.1), we obtain $AC_c^t \oplus \alpha(C'_{local\_i}) \subseteq AC_s^t$.

Combining the results of cases 1, 2 and 3 yields $(AC_c^t \wedge \alpha(C'_{local\_i})) \subseteq AC_s^t$. Moreover, $\alpha(C'_{local\_i}) \subseteq AC_s^n$ and $AC_c^n \oplus \alpha(C'_{local\_i}) \subseteq AC_s^n$ and $AC_c^o \oplus \alpha(C'_{local\_i}) \subseteq AC_s^o$.

4. $\underline{AC_c^o \subseteq AC_s^o}$ (by Definition 5.4.7 of procedure-exit)
5. $\underline{AC_c^n \subseteq AC_s^n}$ (by Definition 5.4.7 of procedure-exit) □

Lemma 5.4.3 shows that the premises of Lemma 5.4.2 hold.

Lemma 5.4.3
Assume the same definitions for Cl_i, (AC_c^o, AC_c^n) and (AC_i^o, AC_i^n) as in Definition 5.4.7. Let $V = \{Y_1, \ldots, Y_k\}$, $W = \{Z_1, \ldots, Z_k\}$ and $\rho = \{Y_1 \leftarrow Z_1, \ldots, Y_k \leftarrow Z_k\}$ (renaming). Let $C_{local\_i} \in SCons$ be the constraint added during execution of Cl_i (projected onto $vars(Cl_i)$) and $C'_{local\_i} = (\exists_W C_{local\_i})\rho^{-1}$.
Then, $(\exists_V AC_c^t)\rho \wedge \alpha(C_{local\_i}) \subseteq AC_i^t$
with $\alpha(C_{local\_i}) \subseteq AC_i^n$ and $(\exists_V AC_c^t)\rho \oplus \alpha(C_{local\_i}) \subseteq AC_i^o$.

PROOF
The proof is by induction on the depth of the proof tree for the call.

Base case : p/k has a proof tree of depth 1, so the i^{th} clause defining p/k contains only a constraint (no subgoals).

The abstract constraint at the beginning of clause Cl_i is $((\overline{\exists}_V AC_c^t)\rho, \emptyset)$ by Definition 5.4.6 of procedure-entry; the abstract constraint at the end is (AC_i^o, AC_i^n) and is obtained by abstract interpretation of the constraint C_{local_i} in the clause. Applying Lemma 5.4.1, $AC_i^t = (\overline{\exists}_V AC_c^t)\rho \wedge \alpha(C_{local\_i})$; moreover, $\alpha(C_{local\_i}) \subseteq AC_i^n$ and $(\overline{\exists}_V AC_c^t)\rho \oplus \alpha(C_{local\_i}) \subseteq AC_i^o$.

Induction : assume that the property to be proved holds for all calls with proof trees of depth $\leq l-1$ and the given call to p/k has a proof tree of depth l ($l > 1$).
Let the clause Cl_i be $p(Z_1, \ldots, Z_k)$:- $G_1, \ldots, G_m$ where each G_j is either a constraint or a subgoal. Let $(AC_{i_0}^o, AC_{i_0}^n) = ((\overline{\exists}_V AC_c^t)\rho, \emptyset)$ be the abstract call constraint at the beginning of Cl_i. Let $(AC_{i_j}^o, AC_{i_j}^n)$ be the abstract success constraint of G_j; $(AC_{i_m}^o, AC_{i_m}^n) = (AC_i^o, AC_i^n)$ is the success constraint at the end of Cl_i.

- For each G_j being a constraint ($G_j \equiv C_j$), it holds that $AC_{i_{j-1}}^t \wedge \alpha(C_j) = AC_{i_j}^t$; moreover, $\alpha(C_j) \subseteq AC_{i_j}^n$ and $AC_{i_{j-1}}^o \oplus \alpha(C_j) \subseteq AC_{i_j}^o$ and $AC_{i_{j-1}}^n \oplus \alpha(C_j) \subseteq AC_{i_j}^n$ and $AC_{i_{j-1}}^o \subseteq AC_{i_j}^o$ and $AC_{i_{j-1}}^n \subseteq AC_{i_j}^n$(due to Lemma 5.4.1).
- For each G_j being a subgoal, let C_j be the set of constraints gathered during execution of G_j (and projected onto the variables of G_j). Since $p(Y_1, \ldots, Y_k)$ has a proof tree of depth l and G_j is a goal in the definition of p/k, G_j has a proof tree of depth $\leq l-1$. The induction hypothesis then states that the property to be proved holds for G_j. So, Lemma 5.4.2 can be applied yielding $AC_{i_{j-1}}^t \wedge \alpha(C_j) \subseteq AC_{i_j}^t$; moreover, $\alpha(C_j) \subseteq AC_{i_j}^n$, $AC_{i_{j-1}}^o \oplus \alpha(C_j) \subseteq AC_{i_j}^o$, $AC_{i_{j-1}}^n \oplus \alpha(C_j) \subseteq AC_{i_j}^n$, $AC_{i_{j-1}}^o \subseteq AC_{i_j}^o$ and $AC_{i_{j-1}}^n \subseteq AC_{i_j}^n$.

So, in any case,

$$AC_{i_{j-1}}^t \wedge \alpha(C_j) \subseteq AC_{i_j}^t \text{ with } \alpha(C_j) \subseteq AC_{i_j}^n,\ AC_{i_{j-1}}^o \oplus \alpha(C_j) \subseteq AC_{i_j}^o,$$
$$AC_{i_{j-1}}^n \oplus \alpha(C_j) \subseteq AC_{i_j}^n,\ AC_{i_{j-1}}^o \subseteq AC_{i_j}^o,\ AC_{i_{j-1}}^n \subseteq AC_{i_j}^n. \tag{5.15}$$

Using a second induction on m it can be shown that

$$AC_{i_0}^t \wedge \alpha(C_1 \wedge \ldots \wedge C_m) \subseteq AC_{i_m}^t \text{ with } \alpha(C_1 \wedge \ldots \wedge C_m) \subseteq AC_{i_m}^n,$$
$$AC_{i_0}^o \oplus \alpha(C_1 \wedge \ldots \wedge C_m) \subseteq AC_{i_m}^o,\ AC_{i_0}^o \subseteq AC_{i_m}^o \text{ and } AC_{i_0}^n \subseteq AC_{i_m}^n. \tag{5.16}$$

base case : $m = 1$
From (5.15) with $j = 1$ we have that $AC_{i_0}^t \wedge \alpha(C_1) \subseteq AC_{i_1}^t$ with $\alpha(C_1) \subseteq AC_{i_1}^n$ and $AC_{i_0}^o \oplus \alpha(C_1) \subseteq AC_{i_1}^o$ and $AC_{i_0}^o \subseteq AC_{i_1}^o$.

induction : if the property holds for 1 up to $m-1$, then it also holds for m

proof of $AC_{i_0}^t \wedge \alpha(C_1 \wedge \ldots \wedge C_m) \subseteq AC_{i_m}^t$

The induction hypothesis states : $AC_{i_0}^t \wedge \alpha(C_1 \wedge \ldots \wedge C_{m-1}) \subseteq AC_{i_{m-1}}^t$.
$AC_{i_0}^t \wedge \alpha(C_1 \wedge \ldots \wedge C_m)$
$\subseteq AC_{i_0}^t \wedge (\alpha(C_1 \wedge \ldots \wedge C_{m-1}) \wedge \alpha(C_m))$
(by Theorem 5.3.1 (safety of $\wedge$) and Proposition 5.3.8 ($\wedge$ preserves order))
$= (AC_{i_0}^t \wedge \alpha(C_1 \wedge \ldots \wedge C_{m-1})) \wedge \alpha(C_m)$ (by Proposition 5.3.4 (associativity of $\wedge$))
$\subseteq AC_{i_{m-1}}^t \wedge \alpha(C_m)$ (by induction hypothesis and Proposition 5.3.8 ($\wedge$ preserves order))
$\subseteq AC_{i_m}^t$ (by (5.15)).

proof of $\alpha(C_1 \wedge \ldots \wedge C_m) \subseteq AC_{i_m}^n$

The induction hypothesis states : $\alpha(C_1 \wedge \ldots \wedge C_{m-1}) \subseteq AC_{i_{m-1}}^n$.

$\alpha(C_1 \wedge \ldots \wedge C_m)$
$\subseteq \alpha(C_1 \wedge \ldots \wedge C_{m-1}) \wedge \alpha(C_m)$ (by Theorem 5.3.1 (safety of $\wedge$))
$= \alpha(C_1 \wedge \ldots \wedge C_{m-1}) \cup \alpha(C_m) \cup \alpha(C_1 \wedge \ldots \wedge C_{m-1}) \oplus \alpha(C_m)$
(by Definition 5.3.1 of $\wedge$)

1. Since $\alpha(C_1 \wedge \ldots \wedge C_{m-1}) \subseteq AC^n_{i_{m-1}}$ (induction hypothesis) and $AC^n_{i_{m-1}} \subseteq AC^n_{i_m}$ (by (5.15)), $\alpha(C_1 \wedge \ldots \wedge C_{m-1}) \subseteq AC^n_{i_m}$.
2. $\alpha(C_m) \subseteq AC^n_{i_m}$ (by (5.15))
3. Since $\alpha(C_1 \wedge \ldots \wedge C_{m-1}) \subseteq AC^n_{i_{m-1}}$ (induction hypothesis) and $\oplus$ preserves order (Proposition 5.3.6), we have that $\alpha(C_1 \wedge \ldots \wedge C_{m-1}) \oplus \alpha(C_m) \subseteq AC^n_{i_{m-1}} \oplus \alpha(C_m)$. Moreover, $AC^n_{i_{m-1}} \oplus \alpha(C_m) \subseteq AC^n_{i_m}$ by (5.15). Hence, $\alpha(C_1 \wedge \ldots \wedge C_{m-1}) \oplus \alpha(C_m) \subseteq AC^n_{i_m}$

Combining 1, 2 and 3 yields $\alpha(C_1 \wedge \ldots \wedge C_m) \subseteq AC^n_{i_m}$.

proof of $AC^o_{i_0} \oplus \alpha(C_1 \wedge \ldots \wedge C_m) \subseteq AC^o_{i_m}$

The induction hypothesis states : $AC^o_{i_0} \oplus \alpha(C_1 \wedge \ldots \wedge C_{m-1}) \subseteq AC^o_{i_{m-1}}$.
$AC^o_{i_0} \oplus \alpha(C_1 \wedge \ldots \wedge C_m)$
$\subseteq AC^o_{i_0} \oplus (\alpha(C_1 \wedge \ldots \wedge C_{m-1}) \wedge \alpha(C_m))$
(by Theorem 5.3.1 (safety of $\wedge$) and Proposition 5.3.6 ($\wedge$ preserves order))
$= AC^o_{i_0} \oplus (\alpha(C_1 \wedge \ldots \wedge C_{m-1}) \cup \alpha(C_m) \cup \alpha(C_1 \wedge \ldots \wedge C_{m-1}) \oplus \alpha(C_m))$
(by Definition 5.3.1 of $\wedge$)
$= [AC^o_{i_0} \oplus \alpha(C_1 \wedge \ldots \wedge C_{m-1})] \cup [AC^o_{i_0} \oplus \alpha(C_m)] \cup [AC^o_{i_0} \oplus (\alpha(C_1 \wedge \ldots \wedge C_{m-1}) \oplus \alpha(C_m))]$
(by Proposition 5.3.1 (distributivity of $\oplus$ w.r.t. $\cup$))

1. $AC^o_{i_0} \oplus \alpha(C_1 \wedge \ldots \wedge C_{m-1}) \subseteq AC^o_{i_{m-1}}$ (induction hypothesis) and $AC^o_{i_{m-1}} \subseteq AC^o_{i_m}$ (by (5.15)). So, $AC^o_{i_0} \oplus \alpha(C_1 \wedge \ldots \wedge C_{m-1}) \subseteq AC^o_{i_m}$.
2. Since $AC^o_{i_0} \subseteq AC^o_{i_{m-1}}$ (by induction hypothesis) and $\oplus$ preserves order (Proposition 5.3.6), $AC^o_{i_0} \oplus \alpha(C_m) \subseteq AC^o_{i_{m-1}} \oplus \alpha(C_m)$. By (5.15), $AC^o_{i_{m-1}} \oplus \alpha(C_m) \subseteq AC^o_{i_m}$. Hence, $AC^o_{i_0} \oplus \alpha(C_m) \subseteq AC^o_{i_m}$.
3. Since $AC^o_{i_0} \oplus \alpha(C_1 \wedge \ldots \wedge C_{m-1}) \subseteq AC^o_{i_{m-1}}$ (induction hypothesis) and $\oplus$ preserves order (Proposition 5.3.6), $(AC^o_{i_0} \oplus \alpha(C_1 \wedge \ldots \wedge C_{m-1})) \oplus \alpha(C_m) \subseteq AC^o_{i_{m-1}} \oplus \alpha(C_m) \subseteq AC^o_{i_m}$ (by (5.15)). By associativity of $\oplus$ (Proposition 5.3.3), also $AC^o_{i_0} \oplus (\alpha(C_1 \wedge \ldots \wedge C_{m-1}) \oplus \alpha(C_m)) \subseteq AC^o_{i_m}$.

Combining 1, 2 and 3 yields $AC^o_{i_0} \oplus \alpha(C_1 \wedge \ldots \wedge C_m) \subseteq AC^o_{i_m}$.

proof of $AC^o_{i_0} \subseteq AC^o_{i_m}$

$AC^o_{i_0} \subseteq AC^o_{i_{m-1}}$ (by induction hypothesis) $\subseteq AC^o_{i_m}$ (by (5.15)).

proof of $AC^n_{i_0} \subseteq AC^n_{i_m}$

$AC^n_{i_0} \subseteq AC^n_{i_{m-1}}$ (by induction hypothesis) $\subseteq AC^n_{i_m}$ (by (5.15)).

Since $AC^t_{i_0} = (\exists_V AC^t_c)\rho$ and $C_1 \wedge \ldots \wedge C_m = C_{local\_i}$ and $(AC^o_{i_m}, AC^n_{i_m}) = (AC^o_i, AC^n_i)$, (5.16) is transformed into $(\exists_V AC^t_c)\rho \wedge \alpha(C_{local\_i}) \subseteq AC^t_i$ with $\alpha(C_{local\_i}) \subseteq AC^n_i$ and $(\exists_V AC^t_c)\rho \oplus \alpha(C_{local\_i}) \subseteq AC^o_i$. □

Proposition 5.4.3 (Safety of procedure-exit)
Let (AC^o_c, AC^n_c) and (AC^o_s, AC^n_s) be resp. the abstract call and success constraint of a procedure call $p(Y_1, \ldots, Y_k)$ Let the head of each clause Cl_i defining p/k be of the form

$p(Z_1,\ldots,Z_k)$ $(1 \leq i \leq m)$. Let $V = \{Y_1,\ldots,Y_k\}$ and $W = \{Z_1,\ldots,Z_k\}$. (AC_i^o, AC_i^n) is the abstract constraint at the end of Cl_i. Let $C_c \in SCons$ such that $\alpha(C_c) \subseteq AC_c^t$ and $C_{in} = (\exists_V C_c)\rho$ with $\rho = \{Y_1 \leftarrow Z_1,\ldots,Y_k \leftarrow Z_k\}$ (renaming). Let $C_{local\_i} \in SCons$ be the constraint added during execution of Cl_i (projected onto $vars(Cl_i)$) and $C'_{local\_i} = (\exists_W C_{local\_i})\rho^{-1}$.
If $\alpha(C_c) \subseteq AC_c^t$ and $\alpha(C_{in} \wedge C_{local\_i}) \subseteq (\exists_V AC_c^t)\rho \wedge \alpha(C_{local\_i})$, then $\alpha(C_c \wedge C'_{local\_i}) \subseteq AC_s^t$.

PROOF

1. Since $\alpha(C_c) \subseteq AC_c^t$ and $\alpha(C'_{local\_i}) \subseteq \alpha(C'_{local\_i})$ it follows from Corollary 5.3.1 (safety of $\wedge$) that $\alpha(C_c \wedge C'_{local\_i}) \subseteq AC_c^t \wedge \alpha(C'_{local\_i})$.
2. By Lemma 5.4.3, $(\exists_V AC_c^t)\rho \wedge \alpha(C_{local\_i}) \subseteq AC_i^t$ with $\alpha(C_{local\_i}) \subseteq AC_i^n$ and $(\exists_V AC_c^t)\rho \oplus \alpha(C_{local\_i}) \subseteq AC_i^o$. This allows to apply Lemma 5.4.2, yielding $AC_c^t \wedge \alpha(C'_{local\_i}) \subseteq AC_s^t$.

Combining 1 and 2 yields that $\alpha(C_c \wedge C'_{local\_i}) \subseteq AC_s^t$. □

5.4.5 Efficiency considerations

5.4.5.1 Time efficiency

As mentioned in Section 5.4.4, the procedure-exit operation recomputes part of the information. An alternative definition of the abstract operations can be formulated which is more efficient but which also incurs a loss of precision. The idea is to split up an abstract constraint in a different way : the combination of information passed down at procedure-entry with local information gathered during the procedure execution is then added to the n-component of a compound abstract constraint rather than to its o-component.

Definition 5.4.8 (Compound abstract constraint (alternative))
A compound abstract constraint AC is either $\bot$ or a pair (AC^o, AC^n) with $AC^o, AC^n \in \wp(\wp_0(Var))$. The component AC^o contains the information passed on at procedure-entry; AC^n contains the information that is obtained from the constraints gathered during local analysis of the procedure body and the combination of it with the information passed down at procedure-entry.

As a consequence, the o-components of compound abstract constraints within the same clause are identical. The definitions of the order relation and least upper bound operation are the same as in the first approach. Also procedure-entry is not changed. The abstract interpretation of a constraint and the procedure-exit operation have to be adapted.

Definition 5.4.9 (Abstract interpretation of a constraint (alternative))
Let $C \in SCons$ and let (AC_c^o, AC_c^n) be the abstract call constraint of C. The abstract success constraint (AC_s^o, AC_s^n) of C is defined as :

1. $AC_s^o = AC_c^o$ *and*
2. $AC_s^n = AC_c^n \cup \alpha(C) \cup (AC_c^t \ominus \alpha(C))$.

Definition 5.4.10 (Procedure-exit (alternative))
Let (AC_c^o, AC_c^n) be the abstract call constraint of a call $p(Y_1, \ldots, Y_k)$. Let the head of each clause Cl_i defining p/k be of the form $p(Z_1, \ldots, Z_k)$ $(1 \leq i \leq m)$. Let (AC_i^o, AC_i^n) be the abstract constraint at the end of Cl_i and $AC_{exit}^n = lub^{\mathcal{F}}(\{\ldots, (\bar{\exists}_{\{Z_1,\ldots,Z_k\}} AC_i^n)\rho^{-1}, \ldots\})$ with $\rho^{-1} = \{Z_1 \leftarrow Y_1, \ldots, Z_k \leftarrow Y_k\}$ (renaming). The abstract success constraint (AC_s^o, AC_s^n) of the call is defined as :

1. $AC_s^o = AC_c^o$
2. $AC_s^n = AC_c^n \cup AC_{exit}^n \cup (compl(\bar{\exists}_{\{Y_1,\ldots,Y_k\}} AC_c^t) \oplus AC_{exit}^n)$
   where $compl(\bar{\exists}_{\{Y_1,\ldots,Y_k\}} AC_c^t) = AC_c^t \setminus \bar{\exists}_{\{Y_1,\ldots,Y_k\}} AC_c^t$.

At procedure-exit, there is no recomputation in contrast with the first approach (Definition 5.4.7). However, precision may be lost. Indeed, AC_{exit}^n contains dependencies due to $(\bar{\exists}_V AC_c^t) \oplus \alpha(C_{local\_i}^t)$. At extension, these dependencies are combined with dependencies in $compl(\bar{\exists}_V AC_c^t)$ (so one performs $(\bar{\exists}_V AC_c^t) \oplus \alpha(C_{local\_i}^t) \oplus compl(\bar{\exists}_V AC_c^t)$). This implies that dependencies in AC_c^t established by different clauses (defining a previous procedure call) are combined where they are in fact independent.

Example 5.4.1
Consider the following program.
? $p(X, Y, T), q(X, Y)$.
$p(X, Y, T) \leftarrow \{X + T = 3\}$.
$p(X, Y, T) \leftarrow \{X - Y = 6\}$.
$q(X, Y) \leftarrow \{X + Y = 2\}$.
The abstract constraint after interpretation of the call $p(X, Y, T)$ is $(\emptyset, \{\{X, T\}, \{X, Y\}\})$. Applying Definition 5.4.10, the abstract constraint after interpretation of the call $q(X, Y)$ is $(\emptyset, \{\{X, T\}, \{X, Y\}, \{X\}, \{Y\}, \{X, Y, T\}, \{Y, T\}, \{T\}\})$. The fact that $\{T\}$ belongs to the success state of q is an overestimation since, at the concrete level, T cannot be non-free after execution of q. Notice that using Definition 5.4.7, the abstract constraint after the call $q(X, Y)$ is $(\emptyset, \{\{X, T\}, \{X, Y\}, \{X\}, \{Y\}, \{X, Y, T\}, \{Y, T\}\})$, which is more precise.

5.4.5.2 Space efficiency

Compound abstract constraints (both as defined in Definition 5.4.1 or 5.4.8) can be reduced by keeping the old and new components disjoint. It means that, at each step in the analysis, the old component is adjusted by removing those sets of variables that also occur within the new component (i.e. $AC'^o = AC^o \setminus AC^n$); the new component itself is not affected. For example, Definition 5.4.5 of abstract interpretation of a constraint then becomes :

Definition 5.4.11 (Abstract interpretation of a constraint (space-optimised))
Let $C \in SCons$ and let (AC_c^o, AC_c^n) be the abstract call constraint of C. The abstract success constraint (AC_s^o, AC_s^n) of C is defined as :

1. $AC_s^n = AC_c^n \wedge \alpha(C)$ *and*
2. $AC_s^o = [AC_c^o \cup (AC_c^o \oplus \alpha(C))] \setminus AC_s^n$.

The definition of the other abstract operations is adjusted in a similar way. The order relation must be redefined as follows : $(AC_1^o, AC_1^n) \leq^{\mathcal{F}} (AC_2^o, AC_2^n)$ iff $AC_1^n \subseteq AC_2^n$ and $AC_1^o \subseteq AC_2^t$.

It can be shown that at each program point AC'^t obtained with the space-optimised definitions is exactly the same as the original AC^t (without space-optimisation of the old components). In other words, no information is lost by performing the optimisation. We briefly outline the argument underlying this statement. Let (AC_1^o, AC_1^n) and (AC_2^o, AC_2^n) be abstract constraints at subsequent program points, so (AC_2^o, AC_2^n) is computed from (AC_1^o, AC_1^n) by adding some[23] $\alpha(C)$. With the original definition of the abstract operations we have that $AC_2^o = AC_1^o \cup (AC_1^o \oplus \alpha(C))$. Let $(AC_1'^o, AC_1^n)$ be the space-optimised version of (AC_1^o, AC_1^n). Then $AC_1'^o \subseteq AC_1^o$ and for each $S \in AC_1^o \setminus AC_1'^o$ holds that $S \in AC_1^n$ (so $AC_1'^t = AC_1^t$). Now, consider the space-optimised abstract constraint $(AC_2'^o, AC_2^n)$ that is computed from $(AC_1'^o, AC_1^n)$ using the space-optimised version of the abstract operations (so $AC_2'^o = [AC_1'^o \cup (AC_1'^o \oplus \alpha(C))] \setminus AC_2^n$). It must be shown that there is no loss of information in $AC_2'^o \cup AC_2^n$ with respect to $AC_2^o \cup AC_2^n$. Two kinds of sets in $AC_2^o \setminus AC_2'^o$ can be distinguished :

- $S \in AC_1^o \setminus AC_1'^o$ or $\in (AC_1^o \setminus AC_1'^o) \oplus \alpha(C)$, i.e. S is computed from a set in $AC_1^o \setminus AC_1'^o$ that was eliminated due to space-optimisation at a previous program point. Such a S is known to be in AC_1^n and, by definition of abstract interpretation of a constraint and procedure-exit, $AC_1^n \subseteq AC_2^n$ and $AC_1^n \oplus \alpha(C) \subseteq AC_2^n$. So, $S \in AC_2^n$.
- $S \in AC_1'^o \cup (AC_1'^o \oplus \alpha(C))$ and $S \in AC_2^n$. Such a S does not occur in $AC_2'^o$ due to local space-optimisation at the current program point.

In each case, $S \in AC_2^n$. Hence $AC_2'^o \cup AC_2^n$ is equal to $AC_2^o \cup AC_2^n$, i.e. $AC_2'^t$ after space-optimisation is equal to the original AC_2^t.
Since the safety propositions of the abstract operations are defined in terms of the AC^t and the AC^t do not change, the propositions remain valid; the lemma's however must be adjusted to take into account the new definition of the AC^o components.

The reduction optimises space but the effect on time efficiency is not clear. The analysis time may improve due to the computation with smaller abstract constraints. However, keeping AC^o and AC^n disjoint incurs a time overhead and potentially implies a decrease of time efficiency. Further investigation is needed.

5.5 Examples

We illustrate the freeness analysis on some program examples.

First of all, consider the following simple program involving only numerical constraints. We assume that the program is called with all variables being free (i.e. starting with the empty constraint store).

?- (AC_0) $Z = 1$, (AC_1) $p(X, Y, Z, T)$ (AC_2).

[23] In between the two program points there is either a constraint, in which case C is this constraint, or a procedure call, in which case C is the constraint gathered during local analysis of the procedure (projected onto the call variables).

$p(X, Y, Z, T) \leftarrow (AC_3)\ X = Y + Z\ (AC_4)$.
$p(X, Y, Z, T) \leftarrow (AC_5)\ X = Y + T\ (AC_6)$.

The compound abstract constraints derived at each program point are the following (using the definitions in Sections 5.4.1, 5.4.2, 5.4.3 and 5.4.4) :

$AC_0 = (\emptyset,\ \emptyset)$
$AC_1 = (\emptyset,\ \{\{Z\}\})$
$AC_2 = (\emptyset,\ \{\{Z\}, \{X, Y, Z\}, \{X, Y\}, \{X, Y, T\}, \{X, Y, Z, T\}\})$

$AC_3 = (\{\{Z\}\},\ \emptyset)$
$AC_4 = (\{\{Z\}, \{X, Y\}\},\ \{\{X, Y, Z\}\})$

$AC_5 = (\{\{Z\}\},\ \emptyset)$
$AC_6 = (\{\{Z\}, \{X, Y, Z, T\}\},\ \{\{X, Y, T\}\})$

All AC_i with $1 \leq i \leq 6$ indicate that Z is possibly non-free. At the end of the first clause for $p/4$ there is a dependency between X and Y obtained via conjunction of $X = Y + Z$ with $Z = 1$ (entailing $X - Y = 1$). Hence, there is a *possible* dependency $\{X, Y\}$ at AC_2. The abstract constraints at each program point cover all linear combinations of the concrete constraints that may occur at that point.
The set $\{Z\}$ in AC_0 is moved from the new to the old component at procedure-entry (cf. AC_3 and AC_5). At procedure-exit, the least upper bound of the new components of AC_4 and AC_6 (i.e. $\{\{X, Y, Z\}, \{X, Y, T\}\}$) is joined with AC_1 to yield AC_2.

The second example involves both numerical and unification constraints. Again, we assume that the program starts off with the empty constraint store.

?- $(AC_0)\ Z = 1,\ (AC_1)\ p(X, Y, Z, T)\ (AC_2)$.

$p(X, Y, Z, T) \leftarrow (AC_3)\ X = f(Z)\ (AC_4)\ , Z + T = 0\ (AC_5)$.
$p(X, Y, Z, T) \leftarrow (AC_6)\ X = g(Y)\ (AC_7)\ , Y - T = 0\ (AC_8)$.

The compound abstract constraints derived at each program point are :

$AC_0 = (\emptyset,\ \emptyset)$
$AC_1 = (\emptyset,\ \{\{Z\}\})$
$AC_2 = (\emptyset,\ \{\{Z\}, \{X\}, \{X, Z\}, \{Z, T\}, \{T\}, \{X, T\}, \{X, Z, T\}, \{X, Y\}, \{X, Y, Z\},$
$\{Y, T\}, \{Y, Z, T\}, \{X, Y, T\}, \{X, Y, Z, T\}\})$

$AC_3 = (\{\{Z\}\},\ \emptyset)$
$AC_4 = (\{\{Z\}, \{X, Z\}, \{X\}\},\ \{\{X\}, \{X, Z\}\})$
$AC_5 = (\{\{Z\}, \{X, Z\}, \{X\}, \{T\}, \{X, T\}, \{X, Z, T\}\},\ \{\{X\}, \{X, Z\}, \{Z, T\}, \{X, Z, T\},$
$\{X, T\}\})$

$AC_6 = (\{\{Z\}\}, \emptyset)$
$AC_7 = (\{\{Z\}, \{X, Z\}, \{X, Y, Z\}\},\ \{\{X\}, \{X, Y\}\})$
$AC_8 = (\{\{Z\}, \{X, Z\}, \{X, Y, Z\}, \{Y, Z, T\}, \{X, Y, Z, T\}, \{X, Z, T\}\},\ \{\{X\}, \{X, Y\},$
$\{Y, T\}, \{X, Y, T\}, \{X, T\}\})$

Space-optimisation would yield (only the changes are mentioned) :

$AC_4^t = (\{\{Z\}\},\ \{\{X\},\{X,Z\}\})$,
$AC_5^t = (\{\{Z\},\{T\}\},\ \{\{X\},\{X,Z\},\{Z,T\},\{X,Z,T\},\{X,T\}\})$.

AC_5 indicates that at the end of the first clause X, Z and T are possibly non-free (they are indeed non-free in the concrete case). At the end of the second clause (cf. AC_8), only X and Z are possibly non-free, which again precisely describes the concrete situation. Hence, at AC_2 where the results of the two clauses are put together, X, Z and T are possibly non-free and only Y is definitely free.

The last example is the well-known *sumlist* program, which again involves mixed constraints. It defines the relation between a list L and the sum S of the list elements. We assume that the program is called with the first argument being possibly non-free and the second argument free.

?- (AC_0) *sumlist*$(List, Sum)$ (AC_1).

sumlist$(L, S) \leftarrow$
(AC_2) $L = [\,]$,
(AC_3) $S = 0$ (AC_4).
sumlist$(L, S) \leftarrow$
(AC_5) $L = [H \mid T]$,
(AC_6) $S = H + S1$,,
(AC_7) *sumlist*$(T, S1)$ (AC_8).

The abstract constraints obtained at each program point are shown below. For simplicity, we only present the AC_i^t instead of showing the components AC_i^o and AC_i^n. Recall that the AC_i^t are sufficient to derive mode and dependency information.

$AC_0^t = \{\{List\}\}$
$AC_1^t = \{\{List\},\{Sum\},\{List, Sum\}\}$

$AC_2^t = \{\{L\}\}$
$AC_3^t = \{\{L\}\}$
$AC_4^t = \{\{L\},\{S\},\{L,S\}\}$

$AC_5^t = \{\{L\}\}$
$AC_6^t = \{\{L\},\{H\},\{T\},\{L,H\},\{L,T\},\{H,T\},\{L,H,T\}\} = \wp_\emptyset(\{L,H,T\})$
$AC_7^t = \{\{L\},\{H\},\{T\},\{L,H\},\{L,T\},\{H,T\},\{L,H,T\},\{S,S1\},\{L,S,S1\},$
$\{H,S,S1\},\{T,S,S1\},\{L,H,S,S1\},\{L,T,S,S1\},\{H,T,S,S1\},$
$\{L,H,T,S,S1\}\}$
$AC_8^t = \wp_\emptyset(\{L,H,T,S,S1\})$

The results show that, after executing *sumlist*/2 with the given call pattern, the second argument is possibly non-free. In the second clause, the possible non-freeness of L in AC_5^t is propagated onto H and T via $L = [H \mid T]$ (cf. the presence of $\{H\}$ and $\{T\}$ in AC_6^t). Just before the recursive call (in AC_7^t) S and $S1$ are still free but there is an (entailed) possible dependency between them.
The call pattern of the recursive call (i.e. AC_7 projected onto the variables T and $S1$ of the call), being $\{\{T\}\}$, is identical (up to renaming) to the original call pattern. So the abstract interpretation procedure immediately reaches a fixpoint.

More examples can be found in Appendix A.

Chapter 6

Minimal freeness abstraction

The freeness abstraction as presented in the previous chapter is limited by its computational complexity, both with respect to space and time. The reason is that abstract constraints tend to become quite large due to an exhaustive enumeration of all possible variable dependencies. In this chapter, we define a more compact freeness abstraction that keeps track of only a minimum of information, rather than representing all information exhaustively. The compressed abstract constraints are safe approximations of the freeness abstract constraints. They allow to reconstruct (possibly a superset of) the original set of possibly non-free variables and variable dependencies.

The first section of this chapter introduces the minimal freeness abstraction. The second section describes the concrete and abstract domain, together with the concretisation and abstraction function. The primitive and higher-level abstract operations are defined in sections three and four. Finally, the minimal freeness analysis is illustrated on some program examples.

In this and the following chapters, the superscript $\mathcal{F}$ refers to the original freeness abstraction (e.g. $\oplus$ of the freeness abstraction is now denoted by $\oplus^{\mathcal{F}}$), while $\mathcal{M}$ refers to the minimal freeness abstraction.

6.1 Introduction

The freeness abstraction presented in the previous chapter explicitly enumerates all possible dependencies between variables, even dependencies that can be obtained by combination (union) of others. The latter are called *non-minimal* dependencies.

Example 6.1.1
Let $C \equiv X + Y = 3 \wedge 2Y + Z = 5$; then $\alpha^{\mathcal{F}}(C) = \{\{X, Y\}, \{Y, Z\}, \{X, Z\}, \{X, Y, Z\}\}$. So C establishes dependencies between pairs of variables, e.g. $\{X, Z\}$ as $Z - 2X = -1$ is entailed, but also the dependency $\{X, Y, Z\}$ as $X + 3Y + Z = 8, 2X + 4Y + Z = 11, \ldots$ are entailed by C. The dependency $\{X, Y, Z\}$ is non-minimal since it can be reconstructed by taking the union of e.g. $\{X, Y\}$ and $\{Y, Z\}$.

The exhaustive enumeration of dependencies in the freeness abstraction contributes to its expressive power and precision, as explained in Section 5.2.3. Recall that, although $\alpha^{\mathcal{F}}(C)$

is closed under union, an abstract constraint is not necessarily closed in general, e.g. the least upper bound operation can give rise to non-closed abstract constraints. Besides the argument of expressivity, the explicit enumeration of dependencies is beneficial to the abstract conjunction since this operation must consider all possible combinations anyway. A disadvantage however is that the freeness abstraction is quite space and time consuming (in the worst case, the size of an abstract constraint is exponential in the number of variables in its domain). A way to reduce its size is to retain only *minimal* sets.

Definition 6.1.1 (Minimal set)
Let $SS \in \wp(\wp_\emptyset(Var))$. Then $S \in SS$ is minimal in SS iff $\nexists S_1, \ldots, S_m \in SS \setminus \{S\}$ $(m \geq 2)$ such that $S = S_1 \cup \ldots \cup S_m$.

Since not all abstract constraints in the freeness abstraction are closed under union, this minimisation may incur some loss of precision/expressivity (although experiments have shown that this rarely occurs in practice). An example will be given further on.

6.2 Concrete and abstract domain

The concrete domain for the minimal freeness abstraction is the same as for the freeness abstraction, i.e. $Con^{\mathcal{C}} = \wp(Cons)$.

The abstract domain, denoted $Con^{\mathcal{M}}$, consists of *minimal abstract constraints*.

Definition 6.2.1 (Minimal abstract constraint)
Let $AC \in \wp(\wp_\emptyset(Var))$. Then AC is minimal iff $\forall S \in AC : S$ is minimal in AC.

Definition 6.2.2 (Abstract domain)
The abstract domain for the minimal freeness abstraction is $Con^{\mathcal{M}} = \{AC \in Con^{\mathcal{F}} \mid AC$ is minimal $\} \cup \{\bot\}$.

As for the freeness abstraction, $\bot$ denotes definite unsatisfiability or unreachability.

The abstraction function $\alpha^{\mathcal{M}}$ can be defined in a straightforward way, using the abstraction function $\alpha^{\mathcal{F}}$ of the freeness domain and the auxiliary function *min* which minimises an abstract constraint of $Con^{\mathcal{F}}$.

Definition 6.2.3 (min)
Let $AC \in Con^{\mathcal{F}}$. Then

$$min(AC) = \begin{cases} \bot & \text{if } AC = \bot \\ \{S \in AC \mid S \text{ is a minimal set in } AC\} & \text{otherwise.} \end{cases}$$

Definition 6.2.4 (Abstraction function)
Let $CS \in Con^{\mathcal{C}}$. Then $\alpha^{\mathcal{M}}(CS) = min(\alpha^{\mathcal{F}}(CS))$.

However, the above definition of $\alpha^{\mathcal{M}}$ can be optimised. Recall that $\alpha^{\mathcal{F}}(CS)$ is defined in terms of the $\alpha^{\mathcal{F}}(C)$ with $C \in CS$. In Definitions 5.2.10, 5.2.12 and 5.2.13 we have $\alpha^{\mathcal{F}}(C) = close(W)$. The closure adds only non-minimal dependencies. So, to obtain $\alpha^{\mathcal{M}}(C)$, only W has to be computed, i.e. $\alpha^{\mathcal{M}}(C) = min(W)$ (note: W itself is not necessarily minimal, cf. Example 6.2.1). Moreover, for a constraint C consisting of *one* primitive constraint, $\alpha^{\mathcal{M}}(C)$ can be computed directly as follows[1]:

1. $\alpha^{\mathcal{M}}(X = Y) = \{\{X, Y\}\}$;
2. $\alpha^{\mathcal{M}}(X = t) = \{\{X\}\} \cup \{\{X, Y\} \mid Y \in vars(t)\}$;
3. $\alpha^{\mathcal{M}}(a_1X_1 + \ldots + a_nX_n \Diamond b) = \{\{X_1, \ldots, X_n\}\}$ where $\Diamond \in \{=, \neq, >, \geq\}$.

The relation between the minimal freeness and the freeness abstraction of a $CS \in Con^{\mathcal{C}}$ is illustrated in Figure 6.1, where $F \in Con^{\mathcal{F}}$ and $M \in Con^{\mathcal{M}}$ such that $F = \alpha^{\mathcal{F}}(CS)$ and $M = \alpha^{\mathcal{M}}(CS)$. The functions *min* and *close* are adjoint functions to go from a $Con^{\mathcal{F}}$ to a $Con^{\mathcal{M}}$ abstraction and vice versa. Closing the $Con^{\mathcal{M}}$ abstraction yields an upper approximation of the $Con^{\mathcal{F}}$ abstraction, i.e. $F \leq^{\mathcal{F}} close(M)$. For the abstraction of a single constraint $C \in Cons$, the relation between F and M is more precise : in that case $F =^{\mathcal{F}} close(M)$; the reason is that the abstraction of a single constraint is closed under union. The relation between $Con^{\mathcal{F}}$ and $Con^{\mathcal{M}}$ can also be expressed in terms of a Galois insertion : $(Con^{\mathcal{F}}, \leq^{\mathcal{F}}) \underset{min}{\overset{close}{\leftrightarrows}} (Con^{\mathcal{M}}, \leq^{\mathcal{M}})$ where $\leq^{\mathcal{M}}$ is defined further on.

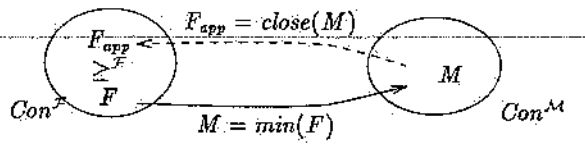


Figure 6.1: Relation between the $Con^{\mathcal{F}}$ and $Con^{\mathcal{M}}$ abstraction

Example 6.2.1
Let $C \equiv X + Y = 4 \wedge Y + Z = 3$. *Then,*

$$\begin{aligned} W &= \{\{X, Y\}, \{Y, Z\}, \{X, Z\}\} \\ \alpha^{\mathcal{M}}(C) &= min(W) = \{\{X, Y\}, \{Y, Z\}, \{X, Z\}\} \end{aligned}$$

Note that $\alpha^{\mathcal{M}}(C)$ *does no longer contain* $\{X, Y, Z\}$ *(which is the abstraction of e.g.* $X + 2Y + Z = 7$ *entailed by* C*), since this set can be constructed by taking the union of e.g.* $\{X, Y\}$ *and* $\{Y, Z\}$. *So* $\alpha^{\mathcal{M}}(C)$ *is more compact than* $\alpha^{\mathcal{F}}(C)$ *(cf. Example 5.2.1).*
Let $C \equiv X = f(Y) \wedge Y = g(U) \wedge U - T - Z = 0$. *Then,*

$$\begin{aligned} W &= \{\{X\}, \{Y\}, \{X, Y\}, \{X, U\}, \\ &\quad \{Y, U\}, \{U, T, Z\}, \{X, T, Z\}, \{Y, T, Z\}\}, \\ \alpha^{\mathcal{M}}(C) &= min(W) = \{\{X\}, \{Y\}, \{X, U\}, \{Y, U\}, \{U, T, Z\}, \\ &\quad \{X, T, Z\}, \{Y, T, Z\}\}. \end{aligned}$$

[1]Primitive constraints are assumed to be in normal form (cf. Section 5.3.3.2), i.e. of the form $X = Y$ or $X = t$ with $t \in T(\Sigma_{\mathcal{H}}, Var)$ and t is not a variable (for unification constraints) or $a_1X_1 + \ldots + a_nX_n \Diamond b$ with $\Diamond \in \{=, \neq, >, \geq\}$ and $a_1, \ldots, a_n, b$ being numbers ($a_i \neq 0$ for i in $1..n$, $a_1 = 1$) (for numerical constraints).

Again, $\alpha^{\mathcal{M}}(C)$ is more compact than the freeness abstraction $\alpha^{\mathcal{F}}(C)$ (cf. Example 5.2.3). Closing the above minimal abstract constraints (in order to go back to the freeness abstraction) yields exactly the freeness abstract constraints, so no precision is lost. However, when considering a constraint set CS that contains more than one constraint, $\alpha^{\mathcal{F}}(CS)$ is not necessarily closed under union and may be more precise than $close(\alpha^{\mathcal{M}}(CS))$, as is shown by the following example: let $CS = \{(X = 1), (Y = Z)\}$; then $\alpha^{\mathcal{F}}(CS) = \alpha^{\mathcal{M}}(CS) = \{\{X\}, \{Y, Z\}\}$, but $close(\alpha^{\mathcal{M}}(CS)) = \{\{X\}, \{Y, Z\}, \{X, Y, Z\}\} >^{\mathcal{F}} \alpha^{\mathcal{F}}(CS)$. Afterwards joining $X = Y$ to CS will yield the set $M = \{\{X\}, \{Y\}, \{Z\}\}$ as $\mathcal{M}$-abstraction, whereas the $\mathcal{F}$-abstraction will be $F = \{\{X\}, \{Y, Z\}, \{Y\}, \{X, Y\}, \{X, Z\}, \{X, Y, Z\}\}$. So, M indicates that all variables are possibly non-free, whereas F implies that only X and Y are possibly non-free and Z is still free. The reason is that the $\mathcal{M}$-abstraction behaves as if the three constraints $X = 1$, $Y = Z$ and $X = Y$ can be combined together (note that $CS' = \{X = 1 \wedge Y = Z\}$ has the same $\mathcal{M}$-abstraction as CS), whereas in the concrete case $X = Y$ is either combined with $X = 1$ or with $Y = Z$ but not with both together. The $\mathcal{F}$-abstraction is able to capture this precisely.

The operations on $Con^{\mathcal{M}}$ can all be defined in terms of the operations on the freeness domain. However, it is often possible to obtain more efficient versions.

Definition 6.2.5 (Abstract computational order)
For every $AC_1, AC_2 \in Con^{\mathcal{M}}$ holds :

$$AC_1 \leq^{\mathcal{M}} AC_2 \quad \textit{iff} \quad AC_1 = \bot \textit{ or } (AC_1 \neq \bot, AC_2 \neq \bot \textit{ and } AC_1 \subseteq close(AC_2)).$$

The condition $AC_1 \subseteq close(AC_2)$ is equivalent to (but can be computed more efficiently than) $close(AC_1) \subseteq close(AC_2)$. The equivalence can be shown as follows :

- $AC_1 \subseteq close(AC_2) \Rightarrow close(AC_1) \subseteq close(AC_2)$:
  $A \in close(AC_1)$ can be written as $A = A_1 \cup \ldots \cup A_m$ $(m \geq 1)$ with each $A_i \in AC_1$. If $AC_1 \subseteq close(AC_2)$, then each $A_i \in close(AC_2)$ and therefore $A = A_1 \cup \ldots \cup A_m \in close(AC_2)$ since $close(AC_2)$ is closed under union.
- $close(AC_1) \subseteq close(AC_2) \Rightarrow AC_1 \subseteq close(AC_2)$:
  follows from $AC_1 \subseteq close(AC_1)$, $close(AC_1) \subseteq close(AC_2)$ and transitivity of $\subseteq$.

The minimal element of $Cons^{\mathcal{M}}$ is $\bot$, the maximal element is $min(\wp_\emptyset(Var)) = \{\{X\} \mid X \in Var\}$.

The equality relation $=^{\mathcal{M}}$ is the following : $AC_1 =^{\mathcal{M}} AC_2$ iff $AC_1 \leq^{\mathcal{M}} AC_2$ and $AC_2 \leq^{\mathcal{M}} AC_1$. This can be simplified to : $AC_1 =^{\mathcal{M}} AC_2$ iff $AC_1 = AC_2$. The reasoning is as follows (not taking into account the trivial $\bot$ case) :

- $AC_1 \leq^{\mathcal{M}} AC_2$ iff $AC_1 \subseteq close(AC_2)$. So for each $S \in AC_1$ holds that $S = T_1 \cup \ldots \cup T_m$ $(m \geq 1)$ with each $T_i \in AC_2$ (by definition of close).
- $AC_2 \leq^{\mathcal{M}} AC_1$ iff $AC_2 \subseteq close(AC_1)$. So for each $T \in AC_2$ holds that $T = S_1 \cup \ldots \cup S_n$ $(n \geq 1)$ with each $S_i \in AC_1$ (by definition of close).

Combining the above yields that each $S \in AC_1$ can be written as $S = T_1 \cup \ldots \cup T_m = (S_{11} \cup \ldots S_{1n_1}) \cup \ldots \cup (S_{m1} \cup \ldots S_{mn_m})$ with each $S_{ij} \in AC_1$. However, as AC_1 is minimal, all S_{ij} and therefore all T_i must be identical and collapse into a single set T. So for each $S \in AC_1$ there exists a $T \in AC_2$ such that $S = T$. Similarly, for each $T \in AC_2$ there exists a $S \in AC_1$ such that $T = S$. Hence $AC_1 = AC_2$.

Definition 6.2.6 (Abstract least upper bound)
Let $AC_1, AC_2 \in Con^{\mathcal{M}}$. *Then* $lub^{\mathcal{M}}(AC_1, AC_2) = min(lub^{\mathcal{F}}(AC_1, AC_2))$.

Notice that $min(lub^{\mathcal{F}}(AC_1, AC_2))$ is equivalent to (but again more efficient than) $min(lub^{\mathcal{F}}(close(AC_1), close(AC_2)))$ (follows immediately from the definitions of $lub^{\mathcal{F}}$ and $lub^{\mathcal{M}}$).

Definition 6.2.7 (Abstract greatest lower bound)
Let $AC_1, AC_2 \in Con^{\mathcal{M}}$. *Then* $glb^{\mathcal{M}}(AC_1, AC_2) = min(glb^{\mathcal{F}}(close(AC_1), close(AC_2)))$.

The $lub^{\mathcal{M}}$ and $glb^{\mathcal{M}}$ operations can easily be generalised to compute the least upper bound and greatest lower bound of n $(n > 2)$ abstract constraints. In the sequel we assume that $lub^{\mathcal{M}}$ and $glb^{\mathcal{M}}$ apply to a set of abstract constraints.

A minimal abstract constraint $AC \in Con^{\mathcal{M}}$ allows to infer mode and dependency information as formalised in Proposition 6.2.1. This is similar to Proposition 5.2.3 for the $\mathcal{F}$ abstraction.

Proposition 6.2.1
Let $AC \in Con^{\mathcal{M}}$. *Then the following holds for each* $CS \in Con^{\mathcal{C}}$ *such that* $\alpha^{\mathcal{M}}(CS) \leq^{\mathcal{M}} AC$:

1. $\{X\} \notin AC$ *implies that* X *is definitely free in each* $C \in CS$. *Alternatively,* $\{X\} \in AC$ *indicates that* X *is* possibly *non-free in some* $C \in CS$.
2. *If* $\{X_1, \ldots, X_n\} \in AC$ *or there exist* $S_1, \ldots, S_m \in AC$ $(m \geq 2)$ *such that* $S_1 \cup \ldots \cup S_m = \{X_1, \ldots, X_n\}$, *then* CS possibly *establishes a dependency between the variables* $X_1, \ldots, X_n$, *i.e. (further) constraining all but one of the variables* may *(further) constrain the remaining variable. Otherwise, there is certainly no dependency* $\{X_1, \ldots, X_n\}$ *in each* $C \in CS$.

Finally, the concretisation function $\gamma^{\mathcal{M}}$ for the minimal freeness analysis is determined by $\alpha^{\mathcal{M}}$ and $\leq^{\mathcal{M}}$.

Definition 6.2.8 (Concretisation function)
Let $AC \in Con^{\mathcal{M}}$. *Then* $\gamma^{\mathcal{M}}(AC) = \bigcup \{CS \in Con^{\mathcal{C}} \mid \alpha^{\mathcal{M}}(CS) \leq^{\mathcal{M}} AC\}$.

Proposition 6.2.3 describes the relation between the concrete domain $Con^{\mathcal{C}}$ and the abstract domain $Con^{\mathcal{M}}$. It is based on the relation between $Con^{\mathcal{C}}$ and $Con^{\mathcal{F}}$ (Proposition 5.2.4) and between $Con^{\mathcal{F}}$ and $Con^{\mathcal{M}}$ (Proposition 6.2.2).

Proposition 6.2.2
$(Con^{\mathcal{F}}, \leq^{\mathcal{F}}) \xleftarrow[min]{close} (Con^{\mathcal{M}}, \leq^{\mathcal{M}})$ *is a Galois insertion.*

PROOF
In order to prove that $(Con^{\mathcal{F}}, \leq^{\mathcal{F}}) \xleftarrow[min]{close} (Con^{\mathcal{M}}, \leq^{\mathcal{M}})$ is a Galois insertion, it has to be shown that

1. $\forall M \in Con^{\mathcal{M}} : min(close(M)) =^{\mathcal{M}} M$; $\forall F \in Con^{\mathcal{F}} : F \leq^{\mathcal{F}} close(min(F))$.
2. *min* and *close* are monotonic.

The proof is as follows :

1. (a) Proof of $\forall M \in Con^{\mathcal{M}} : min(close(M)) =^{\mathcal{M}} M$.
   Follows directly from the fact that M is minimal and the definitions of *close* and *min*.

   (b) Proof of $\forall F \in Con^{\mathcal{F}} : F \leq^{\mathcal{F}} close(min(F))$.
   Follows directly from the definitions of *close* and *min*.

2. (a) *min* is monotonic iff $\forall F_1, F_2 \in Con^{\mathcal{F}} : F_1 \leq^{\mathcal{F}} F_2 \Rightarrow min(F_1) \leq^{\mathcal{M}} min(F_2)$.
   If $F_1 = \bot$ or ($F_2 = \bot$ and therefore also $F_1 = \bot$), the proof is trivial ($\bot$ is the minimal element w.r.t. $\leq^{\mathcal{F}}$ and $\leq^{\mathcal{M}}$).
   Now assume that $F_1 \neq \bot$ and $F_2 \neq \bot$. $F_1 \leq^{\mathcal{F}} F_2$ then means that $F_1 \subseteq F_2$. By definition of *min*, $min(F_1) \subseteq F_1$. By case (1b) above, $F_2 \subseteq close(min(F_2))$. By transitivity of $\subseteq$ we obtain that $min(F_1) \subseteq close(min(F_2))$; so by definition of $\leq^{\mathcal{M}}$, $min(F_1) \leq^{\mathcal{M}} min(F_2)$.

   (b) *close* is monotonic iff $\forall M_1, M_2 \in Con^{\mathcal{M}} : M_1 \leq^{\mathcal{M}} M_2 \Rightarrow close(M_1) \leq^{\mathcal{F}} close(M_2)$.
   If $M_1 = \bot$ or ($M_2 = \bot$ and therefore also $M_1 = \bot$), the proof is trivial.
   Otherwise, $M_1 \leq^{\mathcal{M}} M_2$ means that $M_1 \subseteq close(M_2)$. Then $close(M_1) \subseteq close(M_2)$ based on the following reasoning : $A \in close(M_1)$ can be written as $A = A_1 \cup \ldots \cup A_m$ ($m \geq 1$) with each $A_i \in M_1$; if $M_1 \subseteq close(M_2)$, then each $A_i \in close(M_2)$ and therefore $A = A_1 \cup \ldots \cup A_m \in close(M_2)$ since $close(M_2)$ is closed under union. □

Proposition 6.2.3
$(Con^{\mathcal{C}}, \leq^{\mathcal{C}}) \underset{\alpha^{\mathcal{M}}}{\overset{\gamma^{\mathcal{M}}}{\rightleftarrows}} (Con^{\mathcal{M}}, \leq^{\mathcal{M}})$ *is a Galois insertion.*

PROOF
By Proposition 5.2.4, $(Con^{\mathcal{C}}, \leq^{\mathcal{C}}) \underset{\alpha^{\mathcal{F}}}{\overset{\gamma^{\mathcal{F}}}{\rightleftarrows}} (Con^{\mathcal{F}}, \leq^{\mathcal{F}})$ is known to be a Galois insertion.
By Proposition 6.2.2, $(Con^{\mathcal{F}}, \leq^{\mathcal{F}}) \underset{min}{\overset{close}{\rightleftarrows}} (Con^{\mathcal{M}}, \leq^{\mathcal{M}})$ is a Galois insertion. It is well-known that the composition of two Galois insertions is also a Galois insertion (cf. [25]). So, the composition of the above Galois insertions implies that $(Con^{\mathcal{C}}, \leq^{\mathcal{C}}) \underset{min \circ \alpha^{\mathcal{F}}}{\overset{\gamma^{\mathcal{F}} \circ close}{\rightleftarrows}} (Con^{\mathcal{M}}, \leq^{\mathcal{M}})$ is a Galois insertion, where $min \circ \alpha^{\mathcal{F}} = \alpha^{\mathcal{M}}$ and $\gamma^{\mathcal{F}} \circ close = \gamma^{\mathcal{M}}$. □

6.3 Primitive abstract operations

The primitive abstract operations are abstract conjunction and abstract projection. To simplify the definitions below, we do not explicitly deal with $\bot$ (the treatment of $\bot$ is similar as in case of the freeness abstraction, cf. Section 5.3).

We first define some auxiliary operations.

The $\boxplus$ operation is an optimised version of $\oplus^{\mathcal{F}}$ for use with minimal abstract constraints. It avoids the computation of some sets that are certainly non-minimal and that would therefore be deleted by a subsequent *min* operation (although $\boxplus$ does not exclude *all* non-minimal sets).

Definition 6.3.1 ($\boxplus$)
Let $SS_1, SS_2 \in \wp(\wp_\emptyset(Var))$. Then

$$\begin{aligned} SS_1 \boxplus SS_2 \;=\; & (\,\{\,(A_1 \cup A_2) \setminus D \mid A_1 \in SS_1, A_2 \in SS_2, D = A_1 \cap A_2\} \setminus \{\emptyset\}\,) \cup \\ & \{((A_1 \cup A_2) \setminus D) \cup \{X_i\} \mid A_1 \in SS_1, A_2 \in SS_2, D = A_1 \cap A_2, X_i \in D\} \end{aligned}$$

Let $SS_1 \boxplus_\emptyset SS_2$ be the same as $SS_1 \boxplus SS_2$ but without removal of $\{\emptyset\}$.

It should be clear that $min(SS_1 \boxplus SS_2) = min(SS_1 \oplus^{\mathcal{F}} SS_2)$. So, closing $min(SS_1 \boxplus SS_2)$ under union can produce the same sets as closing $min(SS_1 \oplus^{\mathcal{F}} SS_2)$, and this closure includes all sets in $SS_1 \oplus^{\mathcal{F}} SS_2$.

Procedure-exit and abstract interpretation of a constraint make use of an auxiliary operation $\oplus^{\mathcal{M}}$. This operation can also be used in the definition of abstract conjunction, although a more direct and more efficient definition is possible (cf. further). The idea behind the definition of $\oplus^{\mathcal{M}}$ is to close only the necessary parts of AC_1 and AC_2 (i.e. those parts containing common variables, denoted AC_1^c and AC_2^c), and to avoid as much as possible the generation of non-minimal sets when combining these two parts.

Definition 6.3.2 ($\oplus^{\mathcal{M}}$)
Let $AC_1, AC_2 \in Con^{\mathcal{M}}$. Then $AC_1 \oplus^{\mathcal{M}} AC_2 = min((H \setminus \{\emptyset\}) \cup H_1 \cup H_2)$ with

$$\begin{aligned} H &= close(AC_1^c) \boxplus_\emptyset close(AC_2^c), \\ H_1 &= \{A_1 \cup A_2 \mid A_1 \in AC_1^d, A_2 \in H \cup AC_2\}, \\ H_2 &= \{A_1 \cup A_2 \mid A_1 \in AC_2^d, A_2 \in H \cup AC_1\}^2, \\ AC_1^c &= \{S \in AC_1 \mid vars(S) \cap vars(AC_2) \neq \emptyset\},\ AC_1^d = AC_1 \setminus AC_1^c, \\ AC_2^c &= \{S \in AC_2 \mid vars(S) \cap vars(AC_1) \neq \emptyset\},\ AC_2^d = AC_2 \setminus AC_2^c. \end{aligned}$$

Proposition 6.3.1
Let $AC_1, AC_2 \in Con^{\mathcal{M}}$. Then $close(AC_1 \oplus^{\mathcal{M}} AC_2) = close(AC_1) \oplus^{\mathcal{F}} close(AC_2)$.

PROOF

1. proof of $close(AC_1) \oplus^{\mathcal{F}} close(AC_2) \subseteq close(AC_1 \oplus^{\mathcal{M}} AC_2)$

   Let $S \in close(AC_1) \oplus^{\mathcal{F}} close(AC_2)$ (note : $S \neq \emptyset$ by definition of $\oplus^{\mathcal{F}}$). It must be shown that S can also be obtained as a set or as a union of sets (due to the closure) in $AC_1 \oplus^{\mathcal{M}} AC_2$. For this it is sufficient to show that S can be obtained as a set or as a union of sets in $(H \setminus \{\emptyset\}) \cup H_1 \cup H_2$, since those sets in $(H \setminus \{\emptyset\}) \cup H_1 \cup H_2$ that are no longer in $AC_1 \oplus^{\mathcal{M}} AC_2$ due to the minimisation can be produced as a union of (some of) the minimal sets in $AC_1 \oplus^{\mathcal{M}} AC_2$ anyway.

   S can be written as $S = (S_1 \cup S_2) \setminus T$ with $S_1 \in close(AC_1)$, $S_2 \in close(AC_2)$ and $T \subseteq S_1 \cap S_2$ (note : $S_1 \neq \emptyset$ and $S_2 \neq \emptyset$). As $S_1 \in close(AC_1)$, let

   $$S_1 = \overbrace{A_1^d \cup \ldots \cup A_m^d}^{S_1^d} \cup \overbrace{A_1^c \cup \ldots \cup A_p^c}^{S_1^c} \; (m \geq 0,\ p \geq 0,\ m + p > 0) \qquad (6.1)$$

[2] There is an overlap between H_1 and H_2 as both include $A_1 \cup A_2$ with $A_1 \in AC_1^d$ and $A_2 \in AC_2^d$; however the overlap allows to simplify the formulation.

with each $A_i^d \in AC_1^d$ and each $A_j^c \in AC_1^c$. Similarly,

$$S_2 = \overbrace{B_1^d \cup \ldots \cup B_n^d}^{S_1^d} \cup \overbrace{B_1^c \cup \ldots \cup B_q^c}^{S_1^c} \; (n \geq 0,\ q \geq 0,\ n+q > 0) \qquad (6.2)$$

with each $B_i^d \in AC_2^d$ and each $B_j^c \in AC_2^c$.
So $S = (S_1^d \cup S_1^c \cup S_2^d \cup S_2^c) \setminus T$ where $T \subseteq (S_1^d \cup S_1^c) \cap (S_2^d \cup S_2^c)$; $(S_1^d \cup S_1^c) \cap (S_2^d \cup S_2^c)$ $= S_1^c \cap S_2^c$ as $S_1^d \cap S_2 = \emptyset$ and $S_2^d \cap S_1 = \emptyset$ (by definition of AC_1^c, AC_1^d, AC_2^c and AC_2^d).
Then $S = S_1^d \cup S_2^d \cup (S_1^c \cup S_2^c) \setminus T$ with $T \subseteq S_1^c \cap S_2^c$.
To show that S can also be produced via $AC_1 \oplus^{\mathcal{M}} AC_2$, several cases are distinguished:

a. $W = (S_1^c \cup S_2^c) \setminus T = \emptyset$. This can arise in two situations :
   - $S_1^c = S_2^c = \emptyset$ (note : $S_1^d \neq \emptyset$ and $S_2^d \neq \emptyset$, as $S_1 \neq \emptyset$, $S_2 \neq \emptyset$). Then $S = S_1^d \cup S_2^d$ can be produced as the union of the $A_i^d \cup B_j^d \in H_1$ (also in H_2) with $1 \leq i \leq m$ and $1 \leq j \leq n$ (by (6.1), (6.2) and the definition of H_1).
   - $S_1^c \neq \emptyset$, $S_2^c \neq \emptyset$ and $T = S_1^c \cup S_2^c$. Then $W = \emptyset \in H$ (by definition of $\boxplus_\emptyset$) and S can be obtained as the union of the $A_i^d \cup W \in H_1$ and $B_j^d \cup W \in H_2$ (by (6.1), (6.2) and the definitions of H_1, H_2). Note that if $S_1^d = \emptyset$ and $S_2^d = \emptyset$, then $S = \emptyset$ and $\emptyset$ is removed from H when computing $AC_1 \oplus^{\mathcal{M}} AC_2$; if only $S_1^d = \emptyset$ ($S_2^d \neq \emptyset$), then S can be produced as the union of just the $B_j^d \cup W \in H_2$; similarly, if only $S_2^d = \emptyset$, then S can be produced as the union of just the $A_i^d \cup W \in H_1$.

b. $W = (S_1^c \cup S_2^c) \setminus T \neq \emptyset$. We further distinguish two cases :
   - $S_1^c \neq \emptyset$, $S_2^c \neq \emptyset$.
     Then $W \in close(AC_1^c) \oplus^{\mathcal{F}} close(AC_2^c)$ can be produced as the union of sets in H by definition of $\boxplus_\emptyset$ and the equivalence $min(close(AC_1^c) \oplus^{\mathcal{F}} close(AC_2^c)) = min(close(AC_1^c) \boxplus close(AC_2^c))$; let $W = D_1 \cup \ldots \cup D_k$ with each $D_l \in H$. Then S can be obtained as the union of the $A_i^d \cup D_l \in H_1$ and $B_j^d \cup D_l \in H_2$ (by (6.1), (6.2) and the definitions of H_1, H_2). Note that if $S_1^d = \emptyset$ and $S_2^d = \emptyset$, then S can be obtained via the union of just the $D_l \in H$; if only $S_1^d = \emptyset$ (resp. $S_2^d = \emptyset$, then S can be obtained via the union of just the $B_j^d \cup D_l \in H_2$ (resp. $A_i^d \cup W \in H_1$).
   - either $S_1^c = \emptyset$ or $S_2^c = \emptyset$.
     Assume that $S_1^c = \emptyset$ (then $S_1^d \neq \emptyset$ as $S_1 \neq \emptyset$). Then $S = S_1^d \cup S_2^c \cup S_2^d$ can be obtained as the union of the $A_i^d \cup B_j^c \in H_1$ and $A_i^d \cup B_k^d \in H_1$ (also H_2) (by (6.1) and (6.2)). If $S_2^d = \emptyset$, then S can be obtained as the union of just the $A_i^d \cup B_j^c \in H_1$.
     A similar reasoning can be applied if $S_2^c = \emptyset$.

2. proof of $close(AC_1 \oplus^{\mathcal{M}} AC_2) \subseteq close(AC_1) \oplus^{\mathcal{F}} close(AC_2)$
It should be clear from the definitions of H, H_1 and H_2 that each element of these sets can be written as an element of $close(AC_1) \oplus^{\mathcal{F}} close(AC_2)$. We illustrate this for one particular case, the reasoning is similar for the other cases. For example, consider $S \in H_1$; more precisely, $S = A_1 \cup A_2$ with $A_1 \in AC_1^d$ and $A_2 \in H$. So $A_2 \in close(AC_1^c) \boxplus_\emptyset close(AC_2^c)$; let $A_2 = [(B_1^c \cup \ldots \cup B_m^c) \cup (D_1^c \cup \ldots \cup D_n^c)] \setminus T$ with each $B_i^c \in AC_1^c$, each $D_j^c \in AC_2^c$ and $T \subseteq ((B_1^c \cup \ldots \cup B_m^c) \cap (D_1^c \cup \ldots \cup D_n^c))$. Then

$$\begin{aligned} S &= A_1 \cup A_2 \\ &= A_1 \cup ([(B_1^c \cup \ldots \cup B_m^c) \cup (D_1^c \cup \ldots \cup D_n^c)] \setminus T) \\ &= [(A_1 \cup B_1^c \cup \ldots \cup B_m^c) \cup (D_1^c \cup \ldots \cup D_n^c)] \setminus T) \\ &\qquad \text{with } T \subseteq (A_1 \cup B_1^c \cup \ldots \cup B_m^c) \cap (D_1^c \cup \ldots \cup D_n^c) \\ &\qquad (A_1 \cap (D_1^c \cup \ldots \cup D_n^c) = \emptyset \text{ since } vars(AC_1^d) \cap vars(AC_2) = \emptyset) \end{aligned}$$

Herein, $(A_1 \cup B_1^c \cup \ldots \cup B_m^c) \in close(AC_1)$ and $(D_1^c \cup \ldots \cup D_n^c) \in close(AC_2)$.
So $S \in close(AC_1) \oplus^{\mathcal{F}} close(AC_2)$.

So far we know that each set in $(H \setminus \{\emptyset\}) \cup H_1 \cup H_2$ belongs to $close(AC_1) \oplus^{\mathcal{F}} close(AC_2)$, i.e. $(H \setminus \{\emptyset\}) \cup H_1 \cup H_2 \subseteq close(AC_1) \oplus^{\mathcal{F}} close(AC_2)$. Also, $AC_1 \oplus^{\mathcal{M}} AC_2 \subseteq (H \setminus \{\emptyset\}) \cup H_1 \cup H_2$ by the definition of $AC_1 \oplus^{\mathcal{M}} AC_2$ and the property $min(SS) \subseteq SS$. Combining this yields $AC_1 \oplus^{\mathcal{M}} AC_2 \subseteq close(AC_1) \oplus^{\mathcal{F}} close(AC_2)$. This implies that $close(AC_1 \oplus^{\mathcal{M}} AC_2) \subseteq close(close(AC_1) \oplus^{\mathcal{F}} close(AC_2))$, since *close* preserves the order $\subseteq$ (follows immediately from the definition of *close*). Herein, $close(close(AC_1) \oplus^{\mathcal{F}} close(AC_2)) = close(AC_1) \oplus^{\mathcal{F}} close(AC_2)$ since $close(AC_1) \oplus^{\mathcal{F}} close(AC_2)$ is closed under union. The latter is based on the closedness of $close(AC_1)$ and $close(AC_2)$ and the fact that $\oplus^{\mathcal{F}}$ preserves closedness (Proposition 5.3.5). □

Corollary 6.3.1
$AC_1 \oplus^{\mathcal{M}} AC_2 = min(close(AC_1) \oplus^{\mathcal{F}} close(AC_2))$.

PROOF
By Proposition 6.3.1, $close(AC_1 \oplus^{\mathcal{M}} AC_2) = close(AC_1) \oplus^{\mathcal{F}} close(AC_2)$. This implies that $min(close(AC_1 \oplus^{\mathcal{M}} AC_2)) = min(close(AC_1) \oplus^{\mathcal{F}} close(AC_2))$. Herein, $min(close(AC_1 \oplus^{\mathcal{M}} AC_2)) = AC_1 \oplus^{\mathcal{M}} AC_2$ by the definitions of *min* and *close* and the fact that $AC_1 \oplus^{\mathcal{M}} AC_2$ is minimal. □

An efficient definition of abstract conjunction is obtained by again closing only the necessary parts of AC_1 and AC_2 and by avoiding as much as possible the generation of non-minimal sets when combining the two.

Definition 6.3.3 (Abstract conjunction)
Let $AC_1, AC_2 \in Con^{\mathcal{M}}$. The abstract conjunction, denoted $AC_1 \wedge^{\mathcal{M}} AC_2$, is defined as

$$AC_1 \wedge^{\mathcal{M}} AC_2 = min(\ AC_1 \cup AC_2 \cup [close(AC_1^c) \boxplus close(AC_2^c)]\ )$$

with $AC_1^c = \{S \in AC_1 \mid vars(S) \cap vars(AC_2) \neq \emptyset\}$ *and*
$AC_2^c = \{S \in AC_2 \mid vars(S) \cap vars(AC_1) \neq \emptyset\}$.

The definition of abstract conjunction is equivalent to (but more efficient than) the straightforward definition $AC_1 \wedge^{\mathcal{M}} AC_2 = min(close(AC_1) \wedge^{\mathcal{F}} close(AC_2))$ (or even $AC_1 \wedge^{\mathcal{M}} AC_2 = min(AC_1 \cup AC_2 \cup (AC_1 \oplus^{\mathcal{M}} AC_2))$). The equivalence can be shown in an analogous way as for $\oplus^{\mathcal{M}}$.

Proposition 6.3.2
Let $AC_1, AC_2 \in Con^{\mathcal{M}}$. Then $close(AC_1 \wedge^{\mathcal{M}} AC_2) = close(AC_1) \wedge^{\mathcal{F}} close(AC_2)$.

PROOF Analogous to the proof of Proposition 6.3.1. □

Corollary 6.3.2
Let $AC_1, AC_2 \in Con^{\mathcal{M}}$. Then $AC_1 \wedge^{\mathcal{M}} AC_2 = min(close(AC_1) \wedge^{\mathcal{F}} close(AC_2))$.

PROOF Analogous to the proof of Corollary 6.3.1. □

The safety of abstract conjunction is based on the safety of $\Lambda^{\mathcal{M}}$ with respect to $\Lambda^{\mathcal{F}}$, the safety of $\Lambda^{\mathcal{F}}$ with respect to Λ and the composition of Galois insertions.

Proposition 6.3.3 (Safety of $\Lambda^{\mathcal{M}}$ w.r.t. $\Lambda^{\mathcal{F}}$)
Let $F_1, F_2 \in Con^{\mathcal{F}}$ and $M_1, M_2 \in Con^{\mathcal{M}}$. If $min(F_1) \leq^{\mathcal{M}} M_1$ and $min(F_2) \leq^{\mathcal{M}} M_2$, then $min(F_1\ \Lambda^{\mathcal{F}} F_2) \leq^{\mathcal{M}} M_1\ \Lambda^{\mathcal{M}} M_2$.

PROOF
An alternative formulation of the proposition (using that $(Con^{\mathcal{F}}, \leq^{\mathcal{F}}) \xrightleftharpoons[min]{close} (Con^{\mathcal{M}}, \leq^{\mathcal{M}})$ is a Galois insertion) is : if $F_1 \leq^{\mathcal{F}} close(M_1)$ and $F_2 \leq^{\mathcal{F}} close(M_2)$, then $F_1\ \Lambda^{\mathcal{F}} F_2 \leq^{\mathcal{F}} close(M_1\ \Lambda^{\mathcal{M}} M_2)$.
If $F_1 \leq^{\mathcal{F}} close(M_1)$ and $F_2 \leq^{\mathcal{F}} close(M_2)$, then $F_1\ \Lambda^{\mathcal{F}} F_2 \leq^{\mathcal{F}} close(M_1)\ \Lambda^{\mathcal{F}} close(M_2)$, since $\Lambda^{\mathcal{F}}$ preserves the order $\leq^{\mathcal{F}}$ (Proposition 5.3.8). Then by Proposition 6.3.2, $close(M_1)\ \Lambda^{\mathcal{F}} close(M_2) = close(M_1\ \Lambda^{\mathcal{M}} M_2)$. Hence, $F_1\ \Lambda^{\mathcal{F}} F_2 \leq^{\mathcal{F}} close(M_1\ \Lambda^{\mathcal{M}} M_2)$. □

Theorem 6.3.1 (Safety of abstract conjunction)
Let $CS_1, CS_2 \in Con^{\mathcal{C}}$ and $AC_1, AC_2 \in Con^{\mathcal{M}}$. If $\alpha^{\mathcal{M}}(CS_1) \leq^{\mathcal{M}} AC_1$ and $\alpha^{\mathcal{M}}(CS_2) \leq^{\mathcal{M}} AC_2$, then $\alpha^{\mathcal{M}}(CS_1 \wedge CS_2) \leq^{\mathcal{M}} (AC_1\ \Lambda^{\mathcal{M}} AC_2)$.

PROOF
Follows immediately from

1. the safety of $\Lambda^{\mathcal{F}}$ with respect to $\wedge$ (Corollary 5.3.2) :
   for all $CS_1, CS_2 \in Con^{\mathcal{C}}$ and $F_1, F_2 \in Con^{\mathcal{F}}$, if $\alpha^{\mathcal{F}}(CS_1) \leq^{\mathcal{F}} F_1$ and $\alpha^{\mathcal{F}}(CS_2) \leq^{\mathcal{F}} F_2$, then $\alpha^{\mathcal{F}}(CS_1 \wedge CS_2) \leq^{\mathcal{F}} (F_1\ \Lambda^{\mathcal{F}}\ F_2)$;

2. the safety of $\Lambda^{\mathcal{M}}$ with respect to $\Lambda^{\mathcal{F}}$ (Proposition 6.3.3) :
   for all $F_1, F_2 \in Con^{\mathcal{F}}$ and $AC_1, AC_2 \in Con^{\mathcal{M}}$, if $min(F_1) \leq^{\mathcal{M}} AC_1$ and $min(F_2) \leq^{\mathcal{M}} AC_2$, then $min(F_1\ \Lambda^{\mathcal{F}} F_2) \leq^{\mathcal{M}} AC_1\ \Lambda^{\mathcal{M}} AC_2$;

3. the composition of Galois insertions, where $\alpha^{\mathcal{M}} = min \circ \alpha^{\mathcal{F}}$.

□

The abstract projection operation is defined as in Definition 5.3.2.

Definition 6.3.4 (Abstract projection)
Let $AC \in Con^{\mathcal{M}}$ and $V \subseteq Var$. Then $\exists^{\mathcal{M}}_V AC = \{S \in AC \mid S \subseteq V\}$.

The safety of abstract projection is shown below.

Proposition 6.3.4 (Safety of $\exists^{\mathcal{M}}_V$ w.r.t. $\exists^{\mathcal{F}}_V$)
Let $F \in Con^{\mathcal{F}}$ and $M \in Con^{\mathcal{M}}$. If $min(F) \leq^{\mathcal{M}} M$, then $min(\exists^{\mathcal{F}}_V F) \leq^{\mathcal{M}} \exists^{\mathcal{M}}_V M$.

PROOF
An alternative formulation of the proposition (using that $(Con^{\mathcal{F}}, \leq^{\mathcal{F}}) \xrightarrow[min]{\alpha\text{-}close} (Con^{\mathcal{M}}, \leq^{\mathcal{M}})$ is a Galois insertion) is : if $F \leq^{\mathcal{F}} close(M)$, then $\overline{\exists}_V^{\mathcal{F}} F \leq^{\mathcal{F}} close(\overline{\exists}_V^{\mathcal{M}} M)$.
If $F \leq^{\mathcal{F}} close(M)$, then $\overline{\exists}_V^{\mathcal{F}} F \leq^{\mathcal{F}} \overline{\exists}_V^{\mathcal{F}} close(M)$, since $\overline{\exists}_V^{\mathcal{F}}$ preserves the order $\leq^{\mathcal{F}}$ (Proposition 5.3.12). Herein,

$$\begin{aligned}
\overline{\exists}_V^{\mathcal{F}} close(M) &= \{S \in close(M) \mid S \subseteq V\} \text{ (by Definition 5.3.2 of } \overline{\exists}_V^{\mathcal{F}}) \\
&= \{S \mid S = S_1 \cup \ldots \cup S_n, n \geq 1, S_i \in M\ (1 \leq i \leq n), S \subseteq V\} \\
&\quad \text{(by definition of } close) \\
&= \{S \mid S = S_1 \cup \ldots \cup S_n, n \geq 1, S_i \in \overline{\exists}_V^{\mathcal{M}} M\ (1 \leq i \leq n)\} \\
&\quad \text{(by } S_i \subseteq S \text{ and } S \subseteq V \text{ implies } S_i \subseteq V \text{ and Definition 6.3.4 of } \overline{\exists}_V^{\mathcal{M}}) \\
&= close(\overline{\exists}_V^{\mathcal{M}} M) \text{ (by definition of } close).
\end{aligned}$$

So, if $F \leq^{\mathcal{F}} close(M)$, then $\overline{\exists}_V^{\mathcal{F}} F \leq^{\mathcal{F}} close(\overline{\exists}_V^{\mathcal{M}} M)$. □

Theorem 6.3.2 (Safety of abstract projection)
Let $CS \in Con^{\mathcal{C}}$ and $AC \in Con^{\mathcal{M}}$. If $\alpha^{\mathcal{M}}(CS) \leq^{\mathcal{M}} AC$, then $\alpha^{\mathcal{M}}(\overline{\exists}_V CS) \leq^{\mathcal{M}} \overline{\exists}_V^{\mathcal{M}} AC$.

PROOF
Follows immediately from the safety of $\overline{\exists}_V^{\mathcal{F}}$ with respect to $\overline{\exists}_V$ (Proposition 5.3.3), the safety of $\overline{\exists}_V^{\mathcal{M}}$ with respect to $\overline{\exists}_V^{\mathcal{F}}$ (Proposition 6.3.4) and the composition of Galois insertions (where $\alpha^{\mathcal{M}} = min \circ \alpha^{\mathcal{F}}$). □

6.4 Abstract operations

Again, we do not explicitly deal with the $\bot$ case. Completing the abstract operations to take $\bot$ into account is straightforward (once $\bot$ is obtained at some program point, $\bot$ is propagated to all program points up to the end of the clause).

The safety of all abstract operations can be proved to follow from the safety of abstract conjunction and abstract projection in the same way as for the freeness abstraction and is therefore not worked out here.

6.4.1 Compound abstract constraints

To obtain precise results at procedure-exit, a minimal abstract constraint AC is split up in two components, AC^o and AC^n. It is then called a *compound* minimal abstract constraint (or simply minimal abstract constraint if no confusion is possible).

Definition 6.4.1 (Compound minimal abstract constraint)
A compound minimal abstract constraint AC is either $\bot$ or a pair (AC^o, AC^n) with $AC^o, AC^n \in \wp(\wp(Var))$ and AC^o and AC^n are minimal. The component AC^o contains the information passed on at procedure-entry and the combination of it with the information gathered during local analysis of the procedure body; AC^n contains the information that is obtained from the constraints gathered during local analysis of the procedure body.

Note that AC^o and AC^n are not necessarily disjoint. Also, AC^n may contain sets that are not minimal with respect to AC^o and vice versa.

The abstract domain then consists of *compound* minimal abstract constraints rather than of minimal abstract constraints as such.

Definition 6.4.2 (AC^t)
Let (AC^o, AC^n) be a compound minimal abstract constraint.
Then $AC^t = min(AC^o \cup AC^n)$.

Mode and dependency information is inferred from AC^t.

Definition 6.4.3 (Order)
Let AC_1 and AC_2 be two compound minimal abstract constraints. Then[3]*,*

$$AC_1 \leq^{\mathcal{M}} AC_2 \quad \textit{iff} \quad AC_1 = \bot \textit{ or } (AC_1 = (AC_1^o, AC_1^n), AC_2 = (AC_2^o, AC_2^n) \textit{ and } AC_1^o \subseteq close(AC_2^o) \textit{ and } AC_1^n \subseteq close(AC_2^n)).$$

The equality relation $=^{\mathcal{M}}$ is the following : $AC_1 =^{\mathcal{M}} AC_2$ iff $AC_1 \leq^{\mathcal{M}} AC_2$ and $AC_2 \leq^{\mathcal{M}} AC_1$. Or, after simplification, $AC_1 =^{\mathcal{M}} AC_2$ iff $AC_1 = AC_2$ (i.e. both are $\bot$ or otherwise $AC_1^o = AC_2^o$ and $AC_1^n = AC_2^n$).

Definition 6.4.4 (Least upper bound)
Let AC_1 and AC_2 be two compound minimal abstract constraints. Then,

$$AC_{lub} = lub^{\mathcal{M}}(AC_1, AC_2) = \begin{cases} AC_2 & \textit{if } AC_1 = \bot \\ AC_1 & \textit{if } AC_2 = \bot \\ (min(AC_1^o \cup AC_2^o), min(AC_1^n \cup AC_2^n)) & \textit{otherwise} \end{cases}$$

When using the abstract interpretation system PLAI of Muthukumar and Hermenegildo [96, 94], the above least upper bound operation is only needed when annotating the program: normally only *one* general version of each predicate is given as output. This implies that for each point in a predicate the system has to compute the upper bound of all abstract constraints generated at that point in the different predicate specialisations. During the analysis phase however (cf. Definition 6.4.7 of procedure-exit), only the AC^n_{lub}-component is needed and is computed via Definition 6.2.6.

The greatest lower bound can be defined based on Definition 6.2.7.

6.4.2 Abstract interpretation of a constraint

The abstract interpretation of a constraint C consists of computing the abstraction of C, $\alpha^{\mathcal{M}}(C)$, and joining it with the current abstract constraint store. Hereby, $\alpha^{\mathcal{M}}(C)$ itself and the combination of it with the local information is put into the n-component, whereas the combination with information passed on at procedure-entry is put into the o-component.

[3]The symbol $\leq^{\mathcal{M}}$ is overloaded. Depending on the context, it denotes the order between compound minimal abstract constraints or between minimal abstract constraints as such (Definition 6.2.5).

Definition 6.4.5 (Abstract interpretation of a constraint)
Let $C \in SCons$ and let (AC_c^o, AC_c^n) be the abstract call constraint of C. The abstract success constraint (AC_s^o, AC_s^n) of C is defined as :

1. $AC_s^o = min(AC_c^o \cup (AC_c^o \oplus^{\mathcal{M}} \alpha^{\mathcal{M}}(C)))$ *and*
2. $AC_s^n = AC_c^n \wedge^{\mathcal{M}} \alpha^{\mathcal{M}}(C)$.

6.4.3 Procedure-entry

Procedure-entry is performed in exactly the same way as for the freeness abstraction.

Definition 6.4.6 (Procedure-entry)
Let (AC_c^o, AC_c^n) be the abstract call constraint of a procedure call $p(Y_1, \ldots, Y_k)$ and let $p(Z_1, \ldots, Z_k)$ be the head of a clause[4] used to resolve the call. Then, the abstract call constraint of the clause is defined as $(AC_{in}^o, AC_{in}^n) = (AC_{entry}^o, AC_{entry}^n)\rho$ with $AC_{entry} = (\exists_{\{Y_1,\ldots,Y_k\}} AC_c^i, \emptyset)$ and $\rho = \{Y_1 \leftarrow Z_1, \ldots, Y_k \leftarrow Z_k\}$ (renaming).

6.4.4 Procedure-exit

Procedure-exit computes the abstract success constraint of a procedure call, given its abstract call constraint and the abstract constraints at the end of each clause used to resolve the call.

Definition 6.4.7 (Procedure-exit)
Let (AC_c^o, AC_c^n) be the abstract call constraint of a call $p(Y_1, \ldots, Y_k)$. Let the head of each clause Cl_i defining p/k be of the form $p(Z_1, \ldots, Z_k)$ $(1 \leq i \leq m)$. Let (AC_i^o, AC_i^n) be the abstract constraint at the end of Cl_i and $AC_{exit}^n = lub^{\mathcal{M}}(\{\ldots, (\exists_{\{Z_1,\ldots,Z_k\}} AC_i^n)\rho^{-1}, \ldots\})$ with $lub^{\mathcal{M}}$ defined in Definition 6.2.6 and $\rho^{-1} = \{Z_1 \leftarrow Y_1, \ldots, Z_k \leftarrow Y_k\}$ (renaming). The abstract success constraint (AC_s^o, AC_s^n) of the call is defined as :

1. $AC_s^o = min(AC_c^o \cup (AC_c^o \oplus^{\mathcal{M}} AC_{exit}^n)$
2. $AC_s^n = AC_c^n \wedge^{\mathcal{M}} AC_{exit}^n$

As in the case of the freeness abstraction, part of the information computed during analysis of the procedure body is recomputed at procedure-exit.

6.4.5 Efficiency considerations

6.4.5.1 Time efficiency

To avoid the recomputation at procedure-exit, an alternative definition of a compound minimal abstract constraint (and the associated abstract operations) can be formulated. The idea is the same as in the case of the freeness abstraction (Section 5.4.5.1) : in the alternative definition, the combination of information passed down at procedure-entry with local information gathered during the procedure execution is added to the n-component of a compound abstract constraint rather than to its o-component. The definitions of the abstract operations correspond to the definitions in Section 5.4.5.1 (with α replaced by

[4]The clause is assumed to be renamed apart from $vars(AC_c^i)$.

$\alpha^{\mathcal{M}}$, $\oplus$ by $\oplus^{\mathcal{M}}$, Λ by $\Lambda^{\mathcal{M}}$ and with *min* added whenever the union of minimal sets is computed).
As in the case of the freeness abstraction, this time-efficient approach may cause some loss of precision (cf. Section 5.4.5.1).

6.4.5.2 Space efficiency

Compound minimal abstract constraints (both with respect to Definition 6.4.1 or to the time-efficient definition) can be reduced by ensuring that the component AC^o does not contain sets that appear in AC^n or that are non-minimal with respect to AC^t. It means that, at each step in the analysis, the old component is adjusted as follows : $AC'^o = AC^t \setminus AC^n = min(AC^o \cup AC^n) \setminus AC^n$. For example, the definition of abstract interpretation of a constraint then becomes :

Definition 6.4.8 (Abstract interpretation of a constraint (space-optimised)) *Let $C \in SCons$ and let (AC_c^o, AC_c^n) be the abstract call constraint of C. The abstract success constraint (AC_s^o, AC_s^n) of C is defined as :*

1. $AC_s^n = AC_c^n \ \Lambda^{\mathcal{M}} \ \alpha^{\mathcal{M}}(C)$ *and*
2. $AC_s^o = AC_s^t \setminus AC_s^n$ *with* $AC_s^t = AC_c^t \ \Lambda^{\mathcal{M}} \ \alpha^{\mathcal{M}}(C)$.

The definition of the other abstract operations is adjusted in a similar way. Also, the order relation must be redefined : $(AC_1^o, AC_1^n) \leq^{\mathcal{M}} (AC_2^o, AC_2^n)$ iff $AC_1^n \subseteq close(AC_2^n)$ and $AC_1^o \subseteq close(AC_2^t)$.

It can be shown that at each program point AC'^t obtained with the space-optimised definitions is exactly the same as the original AC^t (without space-optimisation of the old components). In other words, no information is lost by performing the optimisation. We briefly outline the argument underlying this statement. Let (AC_1^o, AC_1^n) and (AC_2^o, AC_2^n) be abstract constraints at subsequent program points, so (AC_2^o, AC_2^n) is computed from (AC_1^o, AC_1^n) by adding some[5] $\alpha(C)$. We know that $AC_2^t = AC_1^t \ \Lambda^{\mathcal{M}} \ \alpha^{\mathcal{M}}(C)$ (due to safety of abstract interpretation of a constraint or safety of procedure-exit). Let $(AC_1'^o, AC_1^n)$ be the space-optimised version of (AC_1^o, AC_1^n). We have that $AC_1'^t = min(AC_1'^o \cup AC_1^n) = min((AC_1^t \setminus AC_1^n) \cup AC_1^n) = AC_1^t$. Then $AC_2'^t = AC_1'^t \ \Lambda^{\mathcal{M}} \ \alpha^{\mathcal{M}}(C)$ by the space-optimised definitions of the abstract operations and, since $AC_1'^t = AC_1^t$, $AC_2'^t = AC_1^t \ \Lambda^{\mathcal{M}} \ \alpha^{\mathcal{M}}(C) = AC_2^t$ (cf. above). So the space-optimised constraint contains the same information as the original one.

The reduction optimises space. Also the effect on time efficiency is expected to be beneficial. The analysis time may decrease due to the computation with smaller abstract constraints. Moreover, in the space-optimised definitions only $\Lambda^{\mathcal{M}}$ is used and no longer $\oplus^{\mathcal{M}}$; the latter is computationally more expensive (cf. Definitions 6.3.2 and 6.3.3). Also, computing with AC^t instead of AC^o may imply computing with a smaller abstract constraint since more minimisation is possible.

[5] In between the two program points there is either a constraint, in which case C is this constraint, or a procedure call, in which case C is the constraint gathered during local analysis of the procedure (projected onto the call variables).

6.5 Examples

We illustrate the minimal freeness analysis on the same program examples as used for the freeness analysis.

First of all, consider the following simple program involving only numerical constraints. We assume that the program is called with all variables being free (i.e. starting with the empty constraint store).

?- $(AC_0)\ Z = 1,\ (AC_1)\ p(X, Y, Z, T)\ (AC_2)$.

$p(X, Y, Z, T) \leftarrow (AC_3)\ X = Y + Z\ (AC_4)$.
$p(X, Y, Z, T) \leftarrow (AC_5)\ X = Y + T\ (AC_6)$.

The compound minimal abstract constraints (not space-optimised) derived at each program point are the following :

$AC_0 = (\emptyset,\ \emptyset)$
$AC_1 = (\emptyset,\ \{\{Z\}\})$
$AC_2 = (\emptyset,\ \{\{Z\}, \{X, Y\}, \{X, Y, T\}\})$
$AC_3 = (\{\{Z\}\},\ \emptyset)$
$AC_4 = (\{\{Z\}, \{X, Y\}\},\ \{\{X, Y, Z\}\})$
$AC_5 = (\{\{Z\}\},\ \emptyset)$
$AC_6 = (\{\{Z\}, \{X, Y, Z, T\}\},\ \{\{X, Y, T\}\})$

Space-optimisation would yield $AC_6' = (\{\{Z\}, \{\{X, Y, T\}\})$ instead of AC_6. All AC_i with $i \geq 1$ indicate that Z is possibly non-free. At the end of the first clause for $p/4$ there is a possible dependency between X and Y obtained via conjunction of $X = Y + Z$ with $Z = 1$ (entailing $X - Y = 1$). The minimal abstract constraint AC_2 is more compact than the corresponding abstract constraint obtained with the freeness analysis; the other abstract constraints are unchanged. With space-optimisation, also AC_6' is more compact than in the $\mathcal{F}$ analysis.

The second example involves both numerical and unification constraints. Again, we assume that the program starts off with the empty constraint store.

?- $(AC_0)\ Z = 1,\ (AC_1)\ p(X, Y, Z, T)\ (AC_2)$.

$p(X, Y, Z, T) \leftarrow (AC_3)\ X = f(Z)\ (AC_4)\ ,\ Z + T = 0\ (AC_5)$.
$p(X, Y, Z, T) \leftarrow (AC_6)\ X = g(Y)\ (AC_7)\ ,\ Y - T = 0\ (AC_8)$.

The compound abstract constraints derived at each program point are :

$AC_0 = (\emptyset,\ \emptyset)$
$AC_1 = (\emptyset,\ \{\{Z\}\})$
$AC_2 = (\emptyset,\ \{\{Z\}, \{X\}, \{T\}, \{X, Y\}, \{Y, T\}\})$
$AC_3 = (\{\{Z\}\},\ \emptyset)$
$AC_4 = (\{\{Z\}, \{X\}\},\ \{\{X\}, \{X, Z\}\})$
$AC_5 = (\{\{Z\}, \{X\}, \{T\}\},\ \{\{X\}, \{X, Z\}, \{Z, T\}, \{X, T\}\})$

$AC_6 = (\{\{Z\}\}, \emptyset)$
$AC_7 = (\{\{Z\},\{X,Z\},\{X,Y,Z\}\}, \{\{X\},\{X,Y\}\})$
$AC_8 = (\{\{Z\},\{X,Z\},\{X,Y,Z\},\{Y,Z,T\},\{X,Z,T\}\}, \{\{X\},\{X,Y\},\{Y,T\},$
$\{X,T\}\})$

Space-optimisation would yield (only the changes are mentioned) :
$AC_4' = (\{\{Z\}\}, \{\{X\},\{X,Z\}\})$ (since $\{X\}$ also occurs in AC^n)
$AC_5' = (\{\{Z\},\{T\}\}, \{\{X\},\{X,Z\},\{Z,T\},\{X,T\}\})$ (since $\{X\}$ also occurs in AC^n)
$AC_7' = (\{\{Z\}\}, \{\{X\},\{X,Y\}\})$ ($\{X,Z\}$ and $\{X,Y,Z\}$ can be produced by $AC_7'^t$)
$AC_8 = (\{\{Z\}\}, \{\{X\},\{X,Y\},\{Y,T\},\{X,T\}\})$

The mode information derivable from the AC_i (or AC_i') is the same as for the freeness abstraction (Section 5.5). No information is lost in the minimisation process. Note however that the minimal freeness abstractions (AC_2, AC_4, AC_5, AC_8) are much more compact than the corresponding freeness abstractions. Space-optimisation causes even more improvement.

The last example is the well-known sumlist program, which again involves mixed constraints. It defines the relation between a list L and the sum S of the list elements. We assume that the program is called with the first argument being possibly non-free and the second argument free.

?- (AC_0) *sumlist*$(List, Sum)$ (AC_1).

sumlist$(L, S) \leftarrow$
(AC_2) $L = [\,]$,
(AC_3) $S = 0$ (AC_4).
sumlist$(L, S) \leftarrow$
(AC_5) $L = [H \mid T]$,
(AC_6) $S = H + S1$,,
(AC_7) *sumlist*$(T, S1)$ (AC_8).

The minimal abstract constraints obtained at each program point are shown below. For simplicity, we only present the AC_i^t instead of showing the components AC_i^o and AC_i^n. Recall that the AC_i^t are sufficient to derive mode and dependency information.

$AC_0^t = \{\{List\}\}$
$AC_1^t = \{\{List\},\{Sum\}\}$

$AC_2^t = \{\{L\}\}$
$AC_3^t = \{\{L\}\}$
$AC_4^t = \{\{L\},\{S\}\}$

$AC_5^t = \{\{L\}\}$
$AC_6^t = \{\{L\},\{H\},\{T\}\}$
$AC_7^t = \{\{L\},\{H\},\{T\},\{S,S1\}\}$
$AC_8^t = \{\{L\},\{H\},\{T\},\{S\},\{S1\}\})$

The results show that, after executing *sumlist*/2 with the given call pattern, the second argument is possibly non-free. In the second clause, the possible non-freeness of L in AC_5^t

is propagated onto H and T via $L = [H \mid T]$ (cf. the presence of $\{H\}$ and $\{T\}$ in AC_6^t). Just before the recursive call (in AC_7^t) S and $S1$ are still free but there is an (entailed) possible dependency between them.
The minimal abstract constraints are much more compact than the abstract constraints obtained with the freeness analysis.

More examples can be found in Appendix A.

Chapter 7

Exploiting definiteness information

In this chapter we describe a second approach – orthogonal to the approach presented in the previous chapter – to reduce the size of the freeness abstract constraints. The idea is to exploit definiteness information [83, 43, 41, 4]. Definite variables, i.e. variables that are constrained to a unique value, contribute in a specific way to the freeness abstraction. The abstract information involving definite variables can easily be extracted from the freeness abstraction and be compressed to the set of definite variables. An abstract constraint is thus split in two parts: one part containing the definite variables and another part containing information on the non-definite variables and their dependencies. In contrast to the minimisation presented in the previous chapter, this approach reduces the freeness abstraction without loss of precision.

In the remainder, the superscript $\mathcal{DF}$ refers to the $\mathcal{DF}$ abstraction, i.e. the freeness abstraction exploiting definiteness information, and $\mathcal{F}$ refers to the original freeness abstraction. The superscript $\mathcal{D}$ refers to the definiteness abstraction, defined by García de la Banda and Hermenegildo in [43, 41]. This abstraction supplies the necessary definiteness information.

The first section of this chapter introduces the $\mathcal{DF}$ abstraction. The second section describes the concrete and abstract domain, together with the concretisation and abstraction function. The primitive and higher-level abstract operations are defined in sections three and four. Finally, the $\mathcal{DF}$ analysis is illustrated on some program examples.

7.1 Introduction

Definiteness information [83, 43, 41, 4] is another form of mode information for CLP programs, complementary to freeness information. A variable C is *definite* in a constraint C iff C constrains X to a unique value. In terms of modes, a definiteness analysis, such as the ones described in [83, 43, 41, 4], derives modes **d** (definite) and **a** (any) whereas a freeness analysis derives modes **f** (free) and **a** (any). The definiteness analysis takes into account *definite* dependencies between variables in order to perform accurate definiteness propagation. We let $(Con^{\mathcal{D}}, \leq^{\mathcal{D}})$ and $\alpha^{\mathcal{D}}$ denote the abstract domain and abstraction function of the definiteness analysis; their definition can be found e.g. in [43, 41].

Definite variables contribute in a specific way to the freeness abstraction. Given the set of definite variables D in $C \in SCons$, $\alpha^{\mathcal{F}}(C)$ can be split into a set of sets containing no definite variables – $compl(D, \alpha^{\mathcal{F}}(C))$ – and a set of sets that do contain definite variables

– $defrelated(D, \alpha^{\mathcal{F}}(C))$. Even if only a subset of all definite variables is known, these variables can be separated out.

Proposition 7.1.1
Let $C \in SCons$. Let D be the set of definite variables in C and $D' \subseteq D$.
Then $\alpha^{\mathcal{F}}(C) = compl(D', \alpha^{\mathcal{F}}(C)) \cup defrelated(D', \alpha^{\mathcal{F}}(C))$ where

- $compl(D', SS) = \{S \in SS \mid S \cap D' = \emptyset\}$ *and*
- $defrelated(D', SS) = \wp_\emptyset(D') \cup \{S_1 \cup S_2 \mid S_1 \in \wp_\emptyset(D'), S_2 \in compl(D', SS)\}$.

PROOF
proof of $compl(D', \alpha^{\mathcal{F}}(C)) \cup defrelated(D', \alpha^{\mathcal{F}}(C)) \subseteq \alpha^{\mathcal{F}}(C)$

1. $compl(D', \alpha^{\mathcal{F}}(C)) \subseteq \alpha^{\mathcal{F}}(C)$ by definition of *compl*.
2. $\wp_\emptyset(D') \subseteq \alpha^{\mathcal{F}}(C)$. Each variable X that is definite in C is by definition also non-free in C, so $\{X\} \in \alpha^{\mathcal{F}}(C)$; since $\alpha^{\mathcal{F}}(C)$ is closed under union any union of these singletons also belongs to $\alpha^{\mathcal{F}}(C)$.
3. $\{S_1 \cup S_2 \mid S_1 \in \wp_\emptyset(D'), S_2 \in compl(D', \alpha^{\mathcal{F}}(C))\} \subseteq \alpha^{\mathcal{F}}(C)$ by the above two cases and the fact that $\alpha^{\mathcal{F}}(C)$ is closed under union.

proof of $\alpha^{\mathcal{F}}(C) \subseteq compl(D', \alpha^{\mathcal{F}}(C)) \cup defrelated(D', \alpha^{\mathcal{F}}(C))$
Let $S \in \alpha^{\mathcal{F}}(C)$. Three cases can be distinguished :

1. $S \cap D' = \emptyset$. Then $S \in compl(D', \alpha^{\mathcal{F}}(C))$ by definition of *compl*.
2. $S \subseteq D'$. Then $S \in \wp_\emptyset(D') \subseteq defrelated(D', \alpha^{\mathcal{F}}(C))$.
3. $S = S' \cup D''$ with $S' \cap D' = \emptyset$ and $D'' \subseteq D'$ (so $D'' \in \wp_\emptyset(D')$). $S \in \alpha^{\mathcal{F}}(C)$ is obtained via the solved form of C (cf. definition of $\alpha^{\mathcal{F}}$); in that solved form, each variable X in D'' is bound to a unique value v and does not occur in any other primitive constraint except $X = v$. So S can only have been put in $\alpha^{\mathcal{F}}(C)$ via the union of S' already in $\alpha^{\mathcal{F}}(C)$ and the union of the singletons $\{X\} \in \alpha^{\mathcal{F}}(C)$ for each $X \in D''$. $S' \in \alpha^{\mathcal{F}}(C)$ and $S' \cap D' = \emptyset$ implies $S' \in compl(D', \alpha^{\mathcal{F}}(C))$. So $S \in \{S_1 \cup S_2 \mid S_1 \in \wp_\emptyset(D'), S_2 \in compl(D', \alpha^{\mathcal{F}}(C))\} \subseteq defrelated(D', \alpha^{\mathcal{F}}(C))$. □

A subset of definite variables, or in other words a safe approximation of the set D of definite variables, is computed by a definiteness analysis. This subset is denoted by $defvars(\alpha^{\mathcal{D}}(C))$ in the sequel. How the interaction between the definiteness and $\mathcal{DF}$ analysis can actually be realised will be discussed in Chapter 10.

Proposition 7.1.1 can easily be lifted to a set of constraints CS where D' is (a subset of) the set of variables that are definite in each $C \in CS$ (so $D' \subseteq \alpha^{\mathcal{D}}(C)$). The underlying argument is that $\alpha^{\mathcal{F}}(CS) = \bigcup_{C \in CS} \alpha^{\mathcal{F}}(C)$ and each $\alpha^{\mathcal{F}}(C)$ can be split into $compl(D', \alpha^{\mathcal{F}}(C)) \cup defrelated(D', \alpha^{\mathcal{F}}(C))$ by Proposition 7.1.1.

The $\mathcal{DF}$ abstraction is based on the observation that the freeness abstraction can be expressed in terms of $compl(D, \alpha^{\mathcal{F}}(CS))$ and D (or also $defvars(\alpha^{\mathcal{D}}(CS))$ instead of D). The $\mathcal{DF}$ abstraction is much more compact than the freeness abstraction. This compact representation incurs no loss of precision, in contrast to the $\mathcal{M}$ abstraction presented in the previous chapter.

7.2 Concrete and abstract domain

The concrete domain for the $\mathcal{DF}$ abstraction is the same as for the freeness abstraction, i.e. $Con^{\mathcal{C}} = \wp(Cons)$. The abstract domain is the following.

Definition 7.2.1 (Abstract domain)
The abstract domain is $Con^{\mathcal{DF}} = \{(D, F^*) \mid D \in \wp(Var), F^* \in \wp(\wp_\emptyset(Var \setminus D))\} \cup \{\bot\}$.

Given a $CS \in Con^{\mathcal{C}}$, the abstraction $\alpha^{\mathcal{DF}}(CS)$ is defined in terms of $\alpha^{\mathcal{F}}(CS)$ and (a subset of) the set of definite variables in CS.

Definition 7.2.2 (Abstraction function)
Let $CS \in Con^{\mathcal{C}}$. *Then* $\alpha^{\mathcal{DF}}(\emptyset) = \bot$, *otherwise* $\alpha^{\mathcal{DF}}(CS) = (D, F^*)$ *where* D *could be given by* $defvars(\alpha^{\mathcal{D}}(CS))$ *and* $F^* = compl(D, \alpha^{\mathcal{F}}(CS))$.

There is an exact correspondence between the $Con^{\mathcal{DF}}$ and $Con^{\mathcal{F}}$ abstractions for a $CS \in Con^{\mathcal{C}}$, as illustrated in Figure 7.1 where $F = \alpha^{\mathcal{F}}(CS)$ and $(D, F^*) = \alpha^{\mathcal{DF}}(CS)$. The function *compl* (defined in Proposition 7.1.1) is used to go from a $Con^{\mathcal{F}}$ to a $Con^{\mathcal{DF}}$ abstraction. Vice versa, the function *extend* transforms a $Con^{\mathcal{DF}}$ abstraction into a $Con^{\mathcal{F}}$ abstraction.

Definition 7.2.3 (extend)
Let $(D, F^*) \in Con^{\mathcal{DF}}$. *Then* $extend(D, F^*) = F^* \cup \wp_\emptyset(D) \cup \{S_1 \cup S_2 \mid S_1 \in \wp_\emptyset(D), S_2 \in F^*\}$.

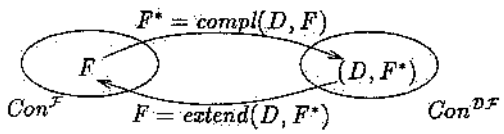


Figure 7.1: Relation between the $Con^{\mathcal{F}}$ and $Con^{\mathcal{DF}}$ abstraction

Example 7.2.1
Let $CS \equiv \{(X + Y + Z = 3 \wedge Y + Z = 2 \wedge T = f(Z)),\ (Y + Z = 5 \wedge T = 1 \wedge X = 3)\}$. *The set of definite variables in* CS *is* $\{X\}$ *as the first constraint in* CS *entails* $X = 1$ *and the second constraint contains* $X = 3$; $\alpha^{\mathcal{F}}(CS) = \{\{X, Y, Z\}, \{X\}, \{Y, Z\}, \{T\}, \{Z, T\}, \{X, T\}, \{Y, T\}, \{X, Z, T\}, \{X, Y, T\}, \{Y, Z, T\}, \{X, Y, Z, T\}\}$. *Then* $\alpha^{\mathcal{DF}}(CS) = (\{X\}, \{\{Y, Z\}, \{T\}, \{Z, T\}, \{Y, T\}, \{Y, Z, T\}\})$. *So* $\alpha^{\mathcal{DF}}(CS)$ *is much more compact than* $\alpha^{\mathcal{F}}(CS)$. *Moreover,* $\alpha^{\mathcal{F}}(CS)$ *can be reconstructed from* $\alpha^{\mathcal{DF}}(CS)$ *via extend, without loss of precision.*

The operations on $Con^{\mathcal{DF}}$ are based on the corresponding operations on $Con^{\mathcal{D}}$ and $Con^{\mathcal{F}}$. To simplify the presentation, we do not explicitly deal with $\bot$. It is straightforward to extend the definitions below, having in mind that $\bot$ is the minimal element in $Con^{\mathcal{DF}}$.

Definition 7.2.4 (Abstract computational order)
Let $(D_1, F_1^*), (D_2, F_2^*) \in Con^{\mathcal{DF}}$.
Then $(D_1, F_1^*) \leq^{\mathcal{DF}} (D_2, F_2^*)$ *iff* $D_1 \supseteq D_2$ *and* $extend(D_1 \setminus D_2, F_1^*) \subseteq F_2^*$.

Note that the above condition is nothing else than $D_1 \leq^{\mathcal{D}} D_2$ and $\mathit{extend}(D_1 \setminus D_2, F_1^*) \leq^{\mathcal{F}} F_2^*$, using[1] $\leq^{\mathcal{D}}$ and $\leq^{\mathcal{F}}$. The minimal element of $Con^{\mathcal{DF}}$ is $\perp$ (denoting definite unsatisfiability or unreachability), the maximal element is $(\emptyset, \wp_\emptyset(Var))$.
The equality relation $=^{\mathcal{DF}}$ induced by $\leq^{\mathcal{DF}}$ is the following : $(D_1, F_1^*) =^{\mathcal{DF}} (D_2, F_2^*)$ iff $D_1 = D_2$ and $F_1^* = F_2^*$.

Definition 7.2.5 (Abstract least upper bound)
Let $(D_1, F_1^*), (D_2, F_2^*) \in Con^{\mathcal{DF}}$. Then $lub^{\mathcal{DF}}((D_1, F_1^*), (D_2, F_2^*)) = (D, F^*)$ with $D = D_1 \cap D_2$ and $F^* = \mathit{extend}(D_1 \setminus (D_1 \cap D_2), F_1^*) \cup \mathit{extend}(D_2 \setminus (D_1 \cap D_2), F_2^*)$.

An alternative formulation is :
$D = lub^{\mathcal{D}}(D_1, D_2)$ and $F = lub^{\mathcal{F}}(\mathit{extend}(D_1 \setminus (D_1 \cap D_2), F_1^*), \mathit{extend}(D_2 \setminus (D_1 \cap D_2), F_2^*))$.
This operation can easily be generalised to compute the least upper bound of n $(n > 2)$ abstract constraints. In the sequel we assume that $lub^{\mathcal{DF}}$ applies to a set of abstract constraints.

An abstract constraint of $Con^{\mathcal{DF}}$ subsumes mode and dependency information as follows.

Proposition 7.2.1
Let $(D, F^*) \in Con^{\mathcal{DF}}$.
Then the following holds for each $CS \in Con^{\mathcal{C}}$ such that $\alpha^{\mathcal{DF}}(CS) \leq^{\mathcal{DF}} (D, F^*)$:

1. *If $X \in D$, then X is a definite variable in CS (mode* d*), else if $\{X\} \notin F^*$ then X is free in CS (mode* f*). Otherwise nothing can be said about X (mode* a*).*
2. *If $\{X_1, \ldots, X_n\} \in F^*$ then CS* possibly *establishes a dependency between the variables $X_1, \ldots, X_n$, i.e. (further) constraining all but one of the variables* may *(further) constrain the remaining variable. Otherwise, there is certainly no dependency $\{X_1, \ldots, X_n\}$ in each $C \in CS$.*

Finally, the concretisation function $\gamma^{\mathcal{DF}}$ is determined by $\alpha^{\mathcal{DF}}$ and $\leq^{\mathcal{DF}}$.

Definition 7.2.6 (Concretisation function)
Let $AC \in Con^{\mathcal{DF}}$. Then $\gamma^{\mathcal{DF}}(AC) = \bigcup \{CS \in Con^{\mathcal{C}} \mid \alpha^{\mathcal{DF}}(CS) \leq^{\mathcal{DF}} AC\}$.

Proposition 7.2.2 describes the relation between the concrete and abstract domain.

Proposition 7.2.2
$(Con^{\mathcal{C}}, \leq^{\mathcal{C}}) \underset{\alpha^{\mathcal{DF}}}{\overset{\gamma^{\mathcal{DF}}}{\rightleftarrows}} (Con^{\mathcal{DF}}, \leq^{\mathcal{DF}})$ is a Galois insertion.

PROOF Similar to the proof of Proposition 5.2.4. □

[1] By abuse of notation, we use $\leq^{\mathcal{D}}$ to compare just a part of the $\mathcal{D}$ abstractions, namely the sets of definite variables, although $\leq^{\mathcal{D}}$ is in fact defined on the entire $\mathcal{D}$ abstractions including also definite dependencies. A similar remark holds for $lub^{\mathcal{D}}$ used further on.

7.3 Primitive abstract operations

The primitive abstract operations are abstract conjunction and abstract projection. For simplicity, we do not explicitly deal with $\bot$ in the definition of the operations.
We first define some auxiliary operations.

Definition 7.3.1 (reduce)
Let $D \in \wp(Var)$ and $F^* \in \wp(\wp_\emptyset(Var))$. Then $reduce(D, F^*) = \{S \setminus D \mid S \in F^*\}$.

Definition 7.3.2 ($\oplus$ w.r.t. D)
Let $D \in \wp(Var)$ and $F_1^*, F_2^* \in \wp(\wp_\emptyset(Var))$. Then $F_1^* \oplus_D F_2^*$ is defined as

$$F_1^* \oplus_D F_2^* = reduce(D, F_1^*) \oplus^{\mathcal{F}} reduce(D, F_2^*).$$

Definition 7.3.3 (Abstract conjunction w.r.t. D)
Let $D \in \wp(Var)$ and $F_1^*, F_2^* \in \wp(\wp_\emptyset(Var))$. The abstract conjunction of F_1^* and F_2^* with respect to D, denoted $F_1^* \ \triangle_D \ F_2^*$, is defined as

$$F_1^* \ \triangle_D \ F_2^* = reduce(D, F_1^*) \ \Lambda^{\mathcal{F}} reduce(D, F_2^*).$$

Definition 7.3.4 (Abstract conjunction)
Let $(D_1, F_1^*), (D_2, F_2^*) \in Con^{\mathcal{DF}}$. Then $(D_1, F_1^*) \ \Lambda^{\mathcal{DF}} \ (D_2, F_2^*) = (D, F^*)$ where D is obtained using the abstract conjunction on $Con^{\mathcal{D}}$ [43, 41] and $F^* = F_1^* \ \triangle_D \ F_2^*$.

D satisfies $D \supseteq D_1 \cup D_2$ (D_1 and D_2 are only parts of the $Con^{\mathcal{D}}$ abstraction, which also contains definite dependencies; so D may be strictly larger than $D_1 \cup D_2$ due to definiteness propagation via these dependencies).

The definition of F^* could be replaced by the following equivalent and straightforward definition : $F^* = compl(D, extend(D_1, F_1^*) \ \Lambda^{\mathcal{F}} \ extend(D_2, F_2^*))$ (cf. Proposition 7.3.1 below). However, computing F^* via $F_1^* \ \triangle_D \ F_2^*$ is more efficient, since the conjunction is performed on smaller abstract constraints. The idea is to join the definiteness parts first and then to propagate the obtained definiteness information (via *reduce*) onto the freeness parts F_1^* and F_2^* before joining them.

Proposition 7.3.1
Let $(D_1, F_1^*), (D_2, F_2^*) \in Con^{\mathcal{DF}}$. Let D be obtained using the abstract conjunction on $Con^{\mathcal{D}}$. Then $F_1^* \ \triangle_D \ F_2^* = compl(D, extend(D_1, F_1^*) \ \Lambda^{\mathcal{F}} extend(D_2, F_2^*))$.

PROOF
By Definition 7.3.3, $F_1^* \ \triangle_D \ F_2^* = reduce(D, F_1^*) \ \Lambda^{\mathcal{F}} reduce(D, F_2^*)$.

proof of $reduce(D, F_1^*) \ \Lambda^{\mathcal{F}} reduce(D, F_2^*) \subseteq compl(D, extend(D_1, F_1^*) \ \Lambda^{\mathcal{F}} extend(D_2, F_2^*))$
The freeness abstraction $extend(D_1, F_1^*) \ \Lambda^{\mathcal{F}} extend(D_2, F_2^*)$ is the abstraction of a set CS of concrete constraints. By safety of the definiteness analysis, $D \subseteq defvars(\alpha^{\mathcal{D}}(CS))$. Then, by Proposition 7.1.1 (lifted to sets of constraints),

$$\begin{aligned} & extend(D_1, F_1^*) \ \Lambda^{\mathcal{F}} extend(D_2, F_2^*) \\ = \ & compl(D, extend(D_1, F_1^*) \ \Lambda^{\mathcal{F}} extend(D_2, F_2^*)) \\ & \cup \ defrelated(D, extend(D_1, F_1^*) \ \Lambda^{\mathcal{F}} extend(D_2, F_2^*)) \end{aligned}$$

As a consequence, for $S \not\subseteq D$ (so $S \setminus D \neq \emptyset$),

$$S \in \mathit{extend}(D_1, F_1^*)\ \Lambda^{\mathcal{F}}\ \mathit{extend}(D_2, F_2^*)$$
$$\text{iff } S \setminus D \in \mathit{extend}(D_1, F_1^*)\ \Lambda^{\mathcal{F}}\ \mathit{extend}(D_2, F_2^*). \tag{7.1}$$

This can be shown as follows, distinguishing two cases for S :

- $S \cap D = \emptyset$:
  Then $S \setminus D = S$. So it is trivial that $S \in \mathit{extend}(D_1, F_1^*)\ \Lambda^{\mathcal{F}}\ \mathit{extend}(D_2, F_2^*)$ iff $S \setminus D \in \mathit{extend}(D_1, F_1^*)\ \Lambda^{\mathcal{F}}\ \mathit{extend}(D_2, F_2^*)$.
- $S \cap D \neq \emptyset$:
  Then $S \notin \mathit{compl}(D, \mathit{extend}(D_1, F_1^*)\ \Lambda^{\mathcal{F}}\ \mathit{extend}(D_2, F_2^*))$.
  So $S \in \mathit{extend}(D_1, F_1^*)\ \Lambda^{\mathcal{F}}\ \mathit{extend}(D_2, F_2^*)$
  iff $S \in \mathit{defrelated}(D, \mathit{extend}(D_1, F_1^*)\ \Lambda^{\mathcal{F}}\ \mathit{extend}(D_2, F_2^*))$
  iff $S \setminus D \in \mathit{compl}(D, \mathit{extend}(D_1, F_1^*)\ \Lambda^{\mathcal{F}}\ \mathit{extend}(D_2, F_2^*))$ (by definition of *defrelated* and the fact that $S \not\subseteq D$)
  iff $S \setminus D \in \mathit{extend}(D_1, F_1^*)\ \Lambda^{\mathcal{F}}\ \mathit{extend}(D_2, F_2^*)$ by $(S \setminus D) \cap D = \emptyset$ and the definition of *compl*.

We now come to the main part of the proof.
By definition of $\Lambda^{\mathcal{F}}$, $\mathit{reduce}(D, F_1^*)\ \Lambda^{\mathcal{F}}\ \mathit{reduce}(D, F_2^*) = \mathit{reduce}(D, F_1^*) \cup \mathit{reduce}(D, F_2^*) \cup (\mathit{reduce}(D, F_1^*) \oplus \mathit{reduce}(D, F_2^*))$. So for $S \in \mathit{reduce}(D, F_1^*)\ \Lambda^{\mathcal{F}}\ \mathit{reduce}(D, F_2^*)$ three cases can be distinguished :

1. $S \in \mathit{reduce}(D, F_1^*)$. Then
   (a1) $S = T \setminus D$ with $T \in F_1^*$ by definition of *reduce*;
   (a2) $T \in \mathit{extend}(D_1, F_1^*)$ since $T \in F_1^*$ (cf. (a1)) and by definition of *extend*;
   (a3) $T \in \mathit{extend}(D_1, F_1^*)\ \Lambda^{\mathcal{F}}\ \mathit{extend}(D_2, F_2^*)$ from (a2) and the definition of $\Lambda^{\mathcal{F}}$;
   (a4) $T \setminus D \in \mathit{extend}(D_1, F_1^*)\ \Lambda^{\mathcal{F}}\ \mathit{extend}(D_2, F_2^*)$ from (a3) and (7.1);
   (a5) $S = T \setminus D \in \mathit{compl}(D, \mathit{extend}(D_1, F_1^*)\ \Lambda^{\mathcal{F}}\ \mathit{extend}(D_2, F_2^*))$ from (a4), the definition of *compl* and the fact that $S \cap D = \emptyset$.

2. $S \in \mathit{reduce}(D, F_2^*)$.
   similar to case 1.

3. $S \in \mathit{reduce}(D, F_1^*)\ \oplus^{\mathcal{F}}\ \mathit{reduce}(D, F_2^*)$. Then
   (b1) $S = (A \cup B) \setminus E$ with $A \in \mathit{reduce}(D, F_1^*)$, $B \in \mathit{reduce}(D, F_2^*)$ and $E \subseteq A \cap B$ by definition of $\oplus^{\mathcal{F}}$; note that $S \cap D = \emptyset$.
   (b2) $A = A' \setminus D$ with $A' \in F_1^*$ and $B = B' \setminus D$ with $B' \in F_2^*$, from (b1) and the definition of *reduce*.
   (b3) $A' \in \mathit{extend}(D_1, F_1^*)$ and $B' \in \mathit{extend}(D_2, F_2^*)$, via (b2) and the definition of *extend*.
   (b4) $A \subseteq A'$ and $B \subseteq B'$ from (b2), so $A \cap B \subseteq A' \cap B'$; combination with (b1) yields $E \subseteq A \cap B \subseteq A' \cap B'$.
   (b5) $(A' \cup B') \setminus E \in \mathit{extend}(D_1, F_1^*)\ \oplus^{\mathcal{F}}\ \mathit{extend}(D_2, F_2^*)$ from (b3), (b4) and the definition of $\oplus^{\mathcal{F}}$; so $(A' \cup B') \setminus E \in \mathit{extend}(D_1, F_1^*)\ \Lambda^{\mathcal{F}}\ \mathit{extend}(D_2, F_2^*)$ by definition of $\Lambda^{\mathcal{F}}$ which implies by (7.1) that $((A' \cup B') \setminus E) \setminus D \in \mathit{extend}(D_1, F_1^*)\ \Lambda^{\mathcal{F}}\ \mathit{extend}(D_2, F_2^*)$.
   (b6) $((A' \cup B') \setminus E) \setminus D$
   $= ((A' \cup B') \setminus D) \setminus E$
   $= (A \cup B) \setminus E$ (via (b2))
   $= S$ (via (b1));

combining this with (b5) yields $S \in extend(D_1, F_1^*)\ \Lambda^{\mathcal{F}} extend(D_2, F_2^*)$ and since $S \cap D = \emptyset$ (cf. (b1)) also $S \in compl(D, extend(D_1, F_1^*)\ \Lambda^{\mathcal{F}} extend(D_2, F_2^*))$.

proof of $compl(D, extend(D_1, F_1^*)\ \Lambda^{\mathcal{F}} extend(D_2, F_2^*)) \subseteq reduce(D, F_1^*)\ \Lambda^{\mathcal{F}} reduce(D, F_2^*)$

$extend(D_1, F_1^*)\ \Lambda^{\mathcal{F}} extend(D_2, F_2^*) = extend(D_1, F_1^*) \cup extend(D_2, F_2^*) \cup (extend(D_1, F_1^*) \oplus^{\mathcal{F}} extend(D_2, F_2^*))$ by definition of $\Lambda^{\mathcal{F}}$. By definition of *compl*, *compl* can be distributed over $\cup$: $compl(D, extend(D_1, F_1^*)\ \Lambda^{\mathcal{F}} extend(D_2, F_2^*)) = compl(D, extend(D_1, F_1^*)) \cup compl(D, extend(D_2, F_2^*)) \cup compl(D, extend(D_1, F_1^*) \oplus^{\mathcal{F}} extend(D_2, F_2^*))$. So three cases are distinguished :

1. $compl(D, extend(D_1, F_1^*)) \subseteq reduce(D, F_1^*) \subseteq reduce(D, F_1^*)\ \Lambda^{\mathcal{F}} reduce(D, F_2^*)$:
   $D \supseteq D_1$ by construction of D (Definition 7.3.4) and the safety of the definiteness analysis. With the definitions of *compl* and *extend* this yields $compl(D, extend(D_1, F_1^*)) \subseteq F_1^*$, so each $S \in compl(D, extend(D_1, F_1^*))$ also belongs to F_1^*. Moreover, for each such a S holds (by definition of *compl*) that $S \cap D = \emptyset$. So, $S \in reduce(D, F_1^*)$.
2. $compl(D, extend(D_2, F_2^*)) \subseteq reduce(D, F_2^*) \subseteq reduce(D, F_1^*)\ \Lambda^{\mathcal{F}} reduce(D, F_2^*)$:
   similar to case 1 (now with $D \supseteq D_2$)
3. $compl(D, extend(D_1, F_1^*) \oplus^{\mathcal{F}} extend(D_2, F_2^*)) \subseteq reduce(D, F_1^*)\ \Lambda^{\mathcal{F}} reduce(D, F_2^*)$:
   Let $S \in compl(D, extend(D_1, F_1^*) \oplus^{\mathcal{F}} extend(D_2, F_2^*))$. Then, by definition of *compl* and $\oplus^{\mathcal{F}}$, $S = (A \cup B) \setminus E$ with $E \subseteq A \cap B$, $A \in extend(D_1, F_1^*)$, $B \in extend(D_2, F_2^*)$ and $S \cap D = \emptyset$. Two cases can be distinguished :
   a. $E \cap D = \emptyset$
      As $S = (A \cup B) \setminus E$ with $S \cap D = \emptyset$ and $E \cap D = \emptyset$, it follows that $A \cap D = \emptyset$ and $B \cap D = \emptyset$. From Definition 7.3.4 of $\Lambda^{\mathcal{DF}}$ and the safety of the definiteness analysis, we have that $D \supseteq D_1$ and $D \supseteq D_2$. Since $D \supseteq D_1$, $A \in extend(D_1, F_1^*)$ and $A \cap D = \emptyset$, it holds that $A \in F_1^*$ and even $\in reduce(D, F_1^*)$ (by definition of reduce). Similarly, since $D \supseteq D_2$, $B \in extend(D_2, F_2^*)$ and $B \cap D = \emptyset$, we have that $B \in F_2^*$ and even $\in reduce(D, F_2^*)$. Hence $S = (A \cup B) \setminus E \in reduce(D, F_1^*) \oplus^{\mathcal{F}} reduce(D, F_2^*)$ by definition of $\oplus^{\mathcal{F}}$ and also $S \in reduce(D, F_1^*)\ \Lambda^{\mathcal{F}} reduce(D, F_2^*)$ by definition of $\Lambda^{\mathcal{F}}$.
   b. $E \cap D \neq \emptyset$
      Let $A = D^A \cup A'$ and $B = D^B \cup B'$ with $D^A = A \cap D$, $D^B = B \cap D$, $A' = A \setminus D^A$ and $B' = B \setminus D^B$. Since $S \cap D = \emptyset$, $D^A \cup D^B \subseteq E$; E can be split into $D^A \cup D^B$ and $E' = E \setminus (D^A \cup D^B)$ where $E' \subseteq A' \cap B'$. Moreover, $(A \cup B) \setminus E = (A' \cup B') \setminus E = (A' \cup B') \setminus E'$. Three cases can be distinguished :
      i. $A' \neq \emptyset$, $B' \neq \emptyset$ (so $A \not\subseteq D$, which implies $A \not\subseteq D_1$ since $D_1 \subseteq D$; also $B \not\subseteq D$ and $B \not\subseteq D_2$)
         It is known that $A \in extend(D_1, F_1^*)$, so $A = D_1^A \cup A''$ with $A'' \in F_1^*$ (note : $A'' \neq \emptyset$ since $A \not\subseteq D_1$). From $A = D^A \cup A'$ and $D_1^A \subseteq D^A$ (since $D_1 \subseteq D$), it follows that $A' \subseteq A''$. More precisely, $A' = A'' \setminus (D^A \setminus D_1^A)$ or also $A' = A'' \setminus D$. Combining $A'' \in F_1^*$ and $A' = A'' \setminus D$ yields that $A' \in reduce(D, F_1^*)$. Similarly, it can be shown that $B' \in reduce(D, F_2^*)$. So $S = (A \cup B) \setminus E = (A' \cup B') \setminus E' \in reduce(D, F_1^*) \oplus^{\mathcal{F}} reduce(D, F_2^*)$ by definition of $\oplus^{\mathcal{F}}$ and also $S \in reduce(D, F_1^*)\ \Lambda^{\mathcal{F}} reduce(D, F_2^*)$ by definition of $\Lambda^{\mathcal{F}}$.
      ii. $A' = \emptyset$ (so $A = D^A$), which implies that $B' \neq \emptyset$ (follows from $S = (D^A \cup D^B \cup B') \setminus E$, $S \cap D = \emptyset$ and $S \neq \emptyset$).
         It is known that $B \in extend(D_2, F_2^*)$, so $B = D_2^B \cup B''$ with $B'' \in F_2^*$ and $B' =$

$B'' \setminus D$ (note : $B'' \neq \emptyset$ as $B' \neq \emptyset$). This implies that $B' \in reduce(D, F_2^*)$ (by definition of *reduce*). Furthermore $S = (A \cup B) \setminus E = (D^A \cup (D^B \cup B')) \setminus E$; as $S \cap D = \emptyset$ and $E \subseteq A \cap B$, we have that $E = D^A = D^B$ and $S = B' \in reduce(D, F_2^*) \subseteq reduce(D, F_1^*)\ \Lambda^{\mathcal{F}} reduce(D, F_2^*)$.

iii. $B' = \emptyset$, which implies that $A' \neq \emptyset$.
analogous to case (ii),
now $S = A' \in reduce(D, F_1^*) \subseteq reduce(D, F_1^*)\ \Lambda^{\mathcal{F}} reduce(D, F_2^*)$. □

The safety of abstract conjunction is based on Proposition 7.3.1 and the safety of $\Lambda^{\mathcal{F}}$. Safety conditions can also be formulated on the separate D and F^* components of the $\mathcal{DF}$ abstractions.

Proposition 7.3.2 (Safety of abstract conjunction)
Let $C_1, C_2 \in SCons$ *and* $(D_1, F_1^*), (D_2, F_2^*), (D, F^*) \in Con^{\mathcal{DF}}$ *with* $(D_1, F_1^*)\ \Lambda^{\mathcal{DF}} (D_2, F_2^*) = (D, F^*)$. *Then*

1. *if* $defvars(\alpha^{\mathcal{D}}(C_1)) \leq^{\mathcal{D}} D_1$ *and* $defvars(\alpha^{\mathcal{D}}(C_2)) \leq^{\mathcal{D}} D_2$,
   then $defvars(\alpha^{\mathcal{D}}(C_1 \wedge C_2)) \leq^{\mathcal{D}} D$.
2. *if* $\alpha^{\mathcal{F}}(C_1) \leq^{\mathcal{F}} extend(D_1, F_1^*)$ *and* $\alpha^{\mathcal{F}}(C_2) \leq^{\mathcal{F}} extend(D_2, F_2^*)$,
   then $\alpha^{\mathcal{F}}(C_1 \wedge C_2) \leq^{\mathcal{F}} extend(D, F^*)$.
3. *if* $\alpha^{\mathcal{DF}}(C_1) \leq^{\mathcal{DF}} (D_1, F_1^*)$ *and* $\alpha^{\mathcal{DF}}(C_2) \leq^{\mathcal{DF}} (D_2, F_2^*)$,
   then $\alpha^{\mathcal{DF}}(C_1 \wedge C_2) \leq^{\mathcal{DF}} (D, F^*)$.

PROOF

1. Cf. safety of the definiteness analysis.
2. Assume that $C_1 \wedge C_2$ is satisfiable; then $\leq^{\mathcal{F}}$ means $\subseteq$.
   We first prove the safety of $\Lambda^{\mathcal{DF}}$ with respect to $\Lambda^{\mathcal{F}}$, i.e. for each $F_1, F_2 \in Con^{\mathcal{F}}$ and $(D_1, F_1^*), (D_2, F_2^*) \in Con^{\mathcal{DF}}$: if $F_1 \leq^{\mathcal{F}} extend(D_1, F_1^*)$ and $F_2 \leq^{\mathcal{F}} extend(D_2, F_2^*)$, then $F_1 \Lambda^{\mathcal{F}} F_2 \leq^{\mathcal{F}} extend(D, F^*)$ where $(D, F^*) = (D_1, F_1^*)\ \Lambda^{\mathcal{DF}} (D_2, F_2^*)$. This proof is as follows :

   $F_1\ \Lambda^{\mathcal{F}} F_2$
   $\leq^{\mathcal{F}} extend(D_1, F_1^*)\ \Lambda^{\mathcal{F}} extend(D_2, F_2^*)$
   (since $F_1 \leq^{\mathcal{F}} extend(D_1, F_1^*)$, $F_2 \leq^{\mathcal{F}} extend(D_2, F_2^*)$ and $\Lambda^{\mathcal{F}}$ preserves the order $\leq^{\mathcal{F}}$ (Proposition 5.3.8))
   $\leq^{\mathcal{F}} extend(D, compl(D, extend(D_1, F_1^*), F_2 \leq^{\mathcal{F}} extend(D_2, F_2^*))))$ by definition of *compl* and *extend*
   $= extend(D, F_1^*\ \triangle_D\ F_2^*)$ (by Proposition 7.3.1)
   $= extend(D, F^*)$ (by definition of F^*)

   The safety of $\Lambda^{\mathcal{DF}}$ with respect to $\wedge$ is then proved by combining this with the safety of $\Lambda^{\mathcal{F}}$ with respect to $\wedge$ (Corollary 5.3.1, stating that for each $C_1, C_2 \in SCons$ and $F_1, F_2 \in Con^{\mathcal{F}}$: if $\alpha^{\mathcal{F}}(C_1) \leq^{\mathcal{F}} F_1$ and $\alpha^{\mathcal{F}}(C_2) \leq^{\mathcal{F}} F_2$, then $\alpha^{\mathcal{F}}(C_1 \wedge C_2) \leq^{\mathcal{F}} F_1\ \Lambda^{\mathcal{F}} F_2$).
3. Let $\alpha^{\mathcal{DF}}(C_1 \wedge C_2) = (D', F'^*)$, i.e. $D' = defvars(\alpha^{\mathcal{D}}(C_1 \wedge C_2))$ and $F'^* = compl(D', \alpha^{\mathcal{F}}(C_1 \wedge C_2))$. Then $(D', F'^*) \leq^{\mathcal{DF}} (D, F^*)$ iff (1) $D' \leq^{\mathcal{D}} D$ (i.e. $D' \supseteq D$) and (2) $extend(D' \setminus D, F'^*) \subseteq F^*$.
   The first part is exactly case 1. The second part is proved by splitting $extend(D' \setminus D, F'^*)$ in three parts, using the definition of *extend* : $extend(D' \setminus D, F'^*) = F'^* \cup \wp_{\emptyset}(D' \setminus D) \cup \{S_1 \cup S_2 \mid S_1 \in \wp_{\emptyset}(D' \setminus D), S_2 \in F'^*\}$.

a. $F'^* = compl(D', \alpha^{\mathcal{F}}(C_1 \wedge C_2))$
$\subseteq compl(D, \alpha^{\mathcal{F}}(C_1 \wedge C_2))$ since $D' \supseteq D$
$\subseteq compl(D, extend(D, F^*))$ since $\alpha^{\mathcal{F}}(C_1 \wedge C_2) \subseteq extend(D, F^*)$ (cf. case 2) and *compl* preserves the order (follows immediately from the definition of *compl*)
$= F^*$ by the definitions of *compl* and *extend* and $D \cap vars(F^*) = \emptyset$.

b. $(D' \setminus D) \subseteq D' = defvars(\alpha^{\mathcal{D}}(C_1 \wedge C_2))$, so $\wp_\emptyset(D' \setminus D) \subseteq \alpha^{\mathcal{F}}(C_1 \wedge C_2)$ since definite variables are also non-free variables. Since $\wp_\emptyset(D' \setminus D) \subseteq \alpha^{\mathcal{F}}(C_1 \wedge C_2)$ and *compl* preserves the order (follows immediately from its definition), we have that $compl(D, \wp_\emptyset(D' \setminus D)) \subseteq compl(D, \alpha^{\mathcal{F}}(C_1 \wedge C_2))$. The left-hand side $compl(D, \wp_\emptyset(D' \setminus D))$ simplifies to $\wp_\emptyset(D' \setminus D)$ by definition of *compl*. For the right-hand side we have that $compl(D, \alpha^{\mathcal{F}}(C_1 \wedge C_2)) \subseteq F^*$ (cf. case 3.a). To conclude, $\wp_\emptyset(D' \setminus D) \subseteq F^*$.

c. For each $S_1 \in \wp_\emptyset(D' \setminus D)$ and each $S_2 \in F'^*$, $S_1 \cup S_2 \in \alpha^{\mathcal{F}}(C_1 \wedge C_2)$ by Proposition 7.1.1; since $S_1 \cup S_2$ does not contain D-variables, $S_1 \cup S_2 \in compl(D, \alpha^{\mathcal{F}}(C_1 \wedge C_2)) \subseteq F^*$ (cf. case 3.a). So $\{S_1 \cup S_2 \mid S_1 \in \wp_\emptyset(D' \setminus D), S_2 \in F'^*\} \subseteq F^*$. □

Definition 7.3.5 (Abstract projection)
Let $(D, F^*) \in Con^{\mathcal{DF}}$ *and* $V \subseteq Var$. *Then* $\overline{\exists}_V^{\mathcal{DF}}(D, F^*) = (D_p, F_p^*)$ *with* $D_p = D \cap V$ *and* $F_p^* = \{S \in F^* \mid S \subseteq V\}$.

Note that one could also write: $D_p = \overline{\exists}_V^{\mathcal{D}} D$ and $F_p^* = \overline{\exists}_V^{\mathcal{F}} F^*$.

Proposition 7.3.3
Let $(D, F^*) \in Con^{\mathcal{DF}}$ *and* $V \subseteq Var$. *Let* $(D_p, F_p^*) = \overline{\exists}_V^{\mathcal{DF}}(D, F^*)$.
Then $extend(D_p, F_p^*) = \overline{\exists}_V^{\mathcal{F}} extend(D, F^*)$.

PROOF
$extend(D, F^*) = F^* \cup \wp_\emptyset(D) \cup \{S_1 \cup S_2 \mid S_1 \in \wp_\emptyset(D), S_2 \in F^*\}$.
So $\overline{\exists}_V^{\mathcal{F}} extend(D, F^*) = \{S \in extend(D, F^*) \mid S \subseteq V\}$
$= \{S \in F^* \mid S \subseteq V\} \cup \{S \in \wp_\emptyset(D) \mid S \subseteq V\} \cup \{S_1 \cup S_2 \mid S_1 \in \wp_\emptyset(D), S_2 \in F^*, S_1 \cup S_2 \subseteq V\}$.
Herein,
1. $\{S \in F^* \mid S \subseteq V\} = F_p^*$;
2. $\{S \in \wp_\emptyset(D) \mid S \subseteq V\} = \wp_\emptyset(D_p)$;
3. $S_1 \cup S_2 \subseteq V$ implies $S_1 \subseteq V$ and $S_2 \subseteq V$, so $\{S_1 \cup S_2 \mid S_1 \in \wp_\emptyset(D), S_2 \in F^*, S_1 \cup S_2 \subseteq V\}$
$= \{S_1 \cup S_2 \mid S_1 \in \wp_\emptyset(D_p), S_2 \in F_p^*\}$.

So, $\overline{\exists}_V^{\mathcal{F}} extend(D, F^*) = F_p^* \cup \wp_\emptyset(D_p) \cup \{S_1 \cup S_2 \mid S_1 \in \wp_\emptyset(D_p), S_2 \in F_p^*\} = extend(D_p, F_p^*)$ by definition of *extend*. □

The safety of abstract projection is based on Proposition 7.3.3 and the safety of abstract projection in the freeness domain $Con^{\mathcal{F}}$. Safety conditions can also be formulated for the separate D and F^* components of the $\mathcal{DF}$ abstractions.

Proposition 7.3.4 (Safety of abstract projection)
Let $C \in SCons$ *and* $(D, F^*), (D_p, F_p^*) \in Con^{\mathcal{DF}}$ *with* $(D_p, F_p^*) = \overline{\exists}_V^{\mathcal{DF}}(D, F^*)$ $(V \subseteq Var)$.
Then
1. *If* $defvars(\alpha^{\mathcal{D}}(C)) \leq^{\mathcal{D}} D$, *then* $defvars(\alpha^{\mathcal{D}}(\overline{\exists}_V C)) \leq^{\mathcal{D}} D_p$.

2. If $\alpha^{\mathcal{F}}(C) \leq^{\mathcal{F}} \mathit{extend}(D, F^*)$, *then* $\alpha^{\mathcal{F}}(\exists_V C) \leq^{\mathcal{F}} \mathit{extend}(D_p, F_p^*)$.
3. If $\alpha^{\mathcal{DF}}(C) \leq^{\mathcal{DF}} (D, F^*)$, *then* $\alpha^{\mathcal{DF}}(\exists_V C) \leq^{\mathcal{DF}} (D_p, F_p^*)$.

PROOF

1. Cf. safety of the definiteness analysis.

2. We first prove the safety of $\overline{\exists}_V^{\mathcal{DF}}$ with respect to $\overline{\exists}_V^{\mathcal{F}}$, i.e. for each $F \in \mathit{Con}^{\mathcal{F}}$ and $(D_p, F_p^*) = \overline{\exists}_V^{\mathcal{DF}}(D, F^*)$: if $F \leq^{\mathcal{F}} \mathit{extend}(D, F^*)$, then $\overline{\exists}_V^{\mathcal{F}} F \leq^{\mathcal{F}} \mathit{extend}(D_p, F_p^*)$. Since $\overline{\exists}_V^{\mathcal{F}}$ preserves the order (Proposition 5.3.12), $F \leq^{\mathcal{F}} (D, F^*)$ implies $\overline{\exists}_V^{\mathcal{F}} F \leq^{\mathcal{F}} \overline{\exists}_V^{\mathcal{F}} \mathit{extend}(D, F^*)$ and $\overline{\exists}_V^{\mathcal{F}} \mathit{extend}(D, F^*) = \mathit{extend}(D_p, F_p^*)$ by Proposition 7.3.3.
The safety of $\overline{\exists}_V^{\mathcal{DF}}$ with respect to $\overline{\exists}_V$ follows from the above and its combination with the safety of $\overline{\exists}_V^{\mathcal{F}}$ with respect to $\overline{\exists}_V$ (for each $C \in \mathit{SCons}$ and $(D, F^*) \in \mathit{Con}^{\mathcal{DF}}$: if $\alpha^{\mathcal{F}}(C) \leq^{\mathcal{F}} F$, then $\alpha^{\mathcal{F}}(\overline{\exists}_V C) \leq^{\mathcal{F}} \overline{\exists}_V^{\mathcal{F}} F$).

3. Let $\alpha^{\mathcal{DF}}(\overline{\exists}_V C) = (D', F'^*)$, so $D' = \mathit{defvars}(\alpha^{\mathcal{D}}(\overline{\exists}_V C))$, $F'^* = \mathit{compl}(D', \alpha^{\mathcal{F}}(\overline{\exists}_V C))$. Then $(D', F'^*) \leq^{\mathcal{DF}} (D_p, F_p^*)$ iff (1) $D' \supseteq D_p$ and (2) $\mathit{extend}(D' \setminus D_p, F'^*) \subseteq F_p^*$.
The first part is exactly case 1. The second part is proved by splitting $\mathit{extend}(D' \setminus D_p, F'^*)$ in three parts, using the definition of *extend* : $\mathit{extend}(D' \setminus D_p, F'^*) = F'^* \cup \wp_\emptyset(D' \setminus D_p) \cup \{S_1 \cup S_2 \mid S_1 \in \wp_\emptyset(D' \setminus D_p), S_2 \in F'^*\}$.

   a. $F'^* = \mathit{compl}(D', \alpha^{\mathcal{F}}(\overline{\exists}_V C))$
   $\subseteq \mathit{compl}(D_p, \alpha^{\mathcal{F}}(\overline{\exists}_V C))$ since $D' \supseteq D_p$
   $\subseteq \mathit{compl}(D_p, \mathit{extend}(D_p, F_p^*))$ since $\alpha^{\mathcal{F}}(\overline{\exists}_V C) \subseteq \mathit{extend}(D_p, F_p^*)$ (cf. case 2) and *compl* preserves the order (follows immediately from the definition of *compl*)
   $= F_p^*$ by the definitions of *compl* and *extend* and $D_p \cap \mathit{vars}(F_p^*) = \emptyset$.

   b. $D' \setminus D_p \subseteq D' = \mathit{defvars}(\alpha^{\mathcal{D}}(\overline{\exists}_V C))$, so $\wp_\emptyset(D' \setminus D_p) \subseteq \alpha^{\mathcal{F}}(\overline{\exists}_V C)$ since definite variables are also non-free variables. Since $\wp_\emptyset(D' \setminus D_p) \subseteq \alpha^{\mathcal{F}}(\overline{\exists}_V C)$ and *compl* preserves the order (follows immediately from its definition), we have that $\mathit{compl}(D_p, \wp_\emptyset(D' \setminus D_p)) \subseteq \mathit{compl}(D_p, \alpha^{\mathcal{F}}(\overline{\exists}_V C))$. The left-hand side $\mathit{compl}(D_p, \wp_\emptyset(D' \setminus D_p))$ simplifies to $\wp_\emptyset(D' \setminus D_p)$ by definition of *compl*. For the right-hand side $\mathit{compl}(D_p, \alpha^{\mathcal{F}}(\overline{\exists}_V C))$, we have that $\mathit{compl}(D_p, \alpha^{\mathcal{F}}(\overline{\exists}_V C)) \subseteq F_p^*$ (cf. case 3.a). To conclude, $\wp_\emptyset(D' \setminus D_p) \subseteq F_p^*$.

   c. For each $S_1 \in \wp_\emptyset(D' \setminus D_p)$ and each $S_2 \in F'^*$, $S_1 \cup S_2 \in \alpha^{\mathcal{F}}(\overline{\exists}_V C)$ by Proposition 7.1.1 and, since $S_1 \cup S_2$ does not contain D_p-variables, $S_1 \cup S_2 \in \mathit{compl}(D_p, \alpha^{\mathcal{F}}(\overline{\exists}_V C)) \subseteq F_p^*$ (cf. case 3.a). So $\{S_1 \cup S_2 \mid S_1 \in \wp_\emptyset(D' \setminus D_p), S_2 \in F'^*\} \subseteq F_p^*$. □

7.4 Abstract operations

For simplicity, we do not consider $\bot$ in the definitions below (they can be completed in a straightforward way).
The safety of the operations depends on the safety of abstract conjunction and abstract projection. It can be proved in a similar way as for the freeness abstraction.

7.4.1 Compound abstract constraints

To obtain precise results at procedure-exit, an abstract constraint AC is split up in two components, AC^o and AC^n. It is then called a *compound* abstract constraint (or simply abstract constraint if no confusion is possible).

Definition 7.4.1 (Compound abstract constraint)
A compound abstract constraint AC is either $\bot$ or $(AC^o, AC^n) = ((D^o, F^{*o}), (D^n, F^{*n}))$ with $D^o, D^n \in \wp(Var)$ and $F^{*o}, F^{*n} \in \wp(\wp_\emptyset(Var \setminus D))$. The component AC^o contains the information passed on at procedure-entry and the combination of it with the information gathered during local analysis of the procedure body; AC^n contains the information that is obtained from the constraints gathered during local analysis of the procedure body.

Property 7.4.1
For each $((D^o, F^{*o}), (D^n, F^{*n}))$ holds that
1. *$compl(D^o \cup D^n, F^{*o}) = F^{*o}$ (i.e. F^{*o} contains no $(D^o \cup D^n)$-variables)*
2. *$compl(D^o \cup D^n, F^{*n}) = F^{*n}$ (i.e. F^{*n} contains no $(D^o \cup D^n)$-variables)*
3. *$D^o \cap D^n = \emptyset$*
4. *F^{*o} and F^{*n} are not necessarily disjoint.*

The abstract domain then consists of compound abstract constraints rather than of abstract constraints as such.

Definition 7.4.2 (AC^t)
Let $(AC^o, AC^n) = ((D^o, F^{*o}), (D^n, F^{*n}))$ be a compound abstract constraint.
Then $AC^t = (D^t, F^{*t}) = (D^o \cup D^n, F^{*o} \cup F^{*n})$.

Mode and dependency information is inferred from AC^t (cf. Proposition 7.2.1).

Definition 7.4.3 (Order relation)
Let AC_1 and AC_2 be two compound abstract constraints. Then[2],

$$((D_1^o, F_1^{*o}), (D_1^n, F_1^{*n})) \leq^{\mathcal{DF}} ((D_2^o, F_2^{*o}), (D_2^n, F_2^{*n})) \text{ iff}$$
$$D_1^o \supseteq D_2^o, D_1^n \supseteq D_2^n,$$
$$extend(D_1^o \setminus D_2^o, F_1^{*o}) \subseteq F_2^{*o} \text{ and } extend(D_1^n \setminus D_2^n, F_1^{*n}) \subseteq F_2^{*n}.$$

This order induces the following equality relation :
$((D_1^o, F_1^{*o}), (D_1^n, F_1^{*n})) =^{\mathcal{DF}} ((D_2^o, F_2^{*o}), (D_2^n, F_2^{*n}))$ iff $D_1^o = D_2^o$, $D_1^n = D_2^n$, $F_1^{*o} = F_2^{*o}$ and $F_1^{*n} = F_2^{*n}$.

Definition 7.4.4 (Least upper bound)
$lub^{\mathcal{DF}}(((D_1^o, F_1^{*o}), (D_1^n, F_1^{*n})), ((D_2^o, F_2^{*o}), (D_2^n, F_2^{*n}))) = ((D_{lub}^o, F_{lub}^{*o}), (D_{lub}^n, F_{lub}^{*n}))$ *with*
1. $D_{lub}^o = (D_1^o \cap D_2^o) \cup (D_1^o \cap D_2^n) \cup (D_1^n \cap D_2^o)$ [3]
2. $D_{lub}^n = (D_1^n \cap D_2^n)$

[2] The symbol $\leq^{\mathcal{DF}}$ is overloaded. Depending on the context, it denotes the order between compound abstract constraints or between plain abstract constraints (Definition 7.2.4).

[3] D_{lub}^o can be computed more efficiently from D_{lub}^t supplied by the definiteness analysis and D_{lub}^n : $D_{lub}^o = D_{lub}^t \setminus D_{lub}^n$.

$$3.\ F^{*o}_{lub} = F^{*o}_1 \cup F^{*o}_2 \cup promote(D^o_1 \setminus D^t_{lub}, F^{*t}_1) \cup promote(D^o_2 \setminus D^t_{lub}, F^{*t}_2) \cup promotecombs(D^n_1 \setminus D^t_{lub}, F^{*o}_1) \cup promotecombs(D^n_2 \setminus D^t_{lub}, F^{*o}_2)$$

$$4.\ F^{*n}_{lub} = F^{*n}_1 \cup F^{*n}_2 \cup promote(D^n_1 \setminus D^t_{lub}, F^{*n}_1) \cup promote(D^n_2 \setminus D^t_{lub}, F^{*n}_2) = extend(D^n_1 \setminus D^t_{lub}, F^{*n}_1) \cup extend(D^n_2 \setminus D^t_{lub}, F^{*n}_2)$$

where $promote(D, F^*) = \wp_\emptyset(D) \cup \{S_1 \cup S_2 \mid S_1 \in \wp_\emptyset(D), S_2 \in F^*\}$ *and* $promotecombs(D, F^*) = \{S_1 \cup S_2 \mid S_1 \in \wp_\emptyset(D), S_2 \in F^*\}$.

According to Definition 7.4.1, only new information or the promotion of new D-information to new F-information should be in F^{*n}_{lub}; old information or the combination of old information with new information must be in F^{*o}_{lub}. The following proposition provides more insight into Definition 7.4.4 and shows that it is compatible with Definition 7.2.5 of the least upper bound on plain abstract constraints.

Proposition 7.4.1
Let $((D^o_1, F^{*o}_1), (D^n_1, F^{*n}_1))$, $((D^o_2, F^{*o}_2), (D^n_2, F^{*n}_2))$ *be two compound abstract constraints and* $((D^o_{lub}, F^{*o}_{lub}), (D^n_{lub}, F^{*n}_{lub})) = lub^{\mathcal{DF}}(((D^o_1, F^{*o}_1), (D^n_1, F^{*n}_1)), ((D^o_2, F^{*o}_2), (D^n_2, F^{*n}_2)))$.
Then $lub^{\mathcal{DF}}((D^t_1, F^{*t}_1), (D^t_2, F^{*t}_2)) = (D^t_{lub}, F^{*t}_{lub})$ *where the latter* $lub^{\mathcal{DF}}$ *is the one of Definition 7.2.5.*

PROOF
$lub^{\mathcal{DF}}((D^t_1, F^{*t}_1), (D^t_2, F^{*t}_2)) = (D^t_{lub}, F^{*t}_{lub})$ with $D^t_{lub} = D^t_1 \cap D^t_2$ and $F^{*t}_{lub} = extend(D^t_1 \setminus D^t_{lub}, F^{*t}_1) \cup extend(D^t_2 \setminus D^t_{lub}, F^{*t}_2)$ by Definition 7.2.5.
It has to be shown that $((D^o_{lub}, F^{*o}_{lub}), (D^n_{lub}, F^{*n}_{lub}))$ obtained via Definition 7.4.4 yields the same D^t_{lub} and F^{*t}_{lub}.

$D^t_{lub} = D^o_{lub} \cup D^n_{lub}$ by Definition 7.4.2
$= (D^o_1 \cap D^o_2) \cup (D^o_1 \cap D^n_2) \cup (D^n_1 \cap D^o_2) \cup (D^n_1 \cap D^n_2)$
by Definition 7.4.4 of $lub^{\mathcal{DF}}$ on compound abstract constraints
$= (D^o_1 \cup D^n_1) \cap (D^o_2 \cup D^n_2)$ by distributivity of $\cap$ over $\cup$
$= D^t_1 \cap D^t_2$ by Definition 7.4.2

This corresponds with the definition of D^t_{lub} in Definition 7.2.5.

$F^{*t}_{lub} = F^{*o}_{lub} \cup F^{*n}_{lub}$ by Definition 7.4.2
$= F^{*o}_1 \cup F^{*o}_2 \cup promote(D^o_1 \setminus D^t_{lub}, F^{*t}_1) \cup promote(D^o_2 \setminus D^t_{lub}, F^{*t}_2) \cup promotecombs(D^n_1 \setminus D^t_{lub}, F^{*o}_1) \cup promotecombs(D^n_2 \setminus D^t_{lub}, F^{*o}_2) \cup F^{*n}_1 \cup F^{*n}_2 \cup promote(D^n_1 \setminus D^t_{lub}, F^{*n}_1) \cup promote(D^n_2 \setminus D^t_{lub}, F^{*n}_2)$
by Definition 7.4.4 of $lub^{\mathcal{DF}}$ on compound abstract constraints

Herein,
$promotecombs(D^n_1 \setminus D^t_{lub}, F^{*o}_1) \cup promote(D^n_1 \setminus D^t_{lub}, F^{*n}_1) = promote(D^n_1 \setminus D^t_{lub}, F^{*t}_1)$ by the definitions of *promote* and *promotecombs* and Definition 7.4.2.
$promotecombs(D^n_2 \setminus D^t_{lub}, F^{*o}_2) \cup promote(D^n_2 \setminus D^t_{lub}, F^{*n}_2) = promote(D^n_2 \setminus D^t_{lub}, F^{*t}_2)$ by the definitions of *promote* and *promotecombs* and Definition 7.4.2.
Then,
$promote(D^n_1 \setminus D^t_{lub}, F^{*t}_1) \cup promote(D^o_1 \setminus D^t_{lub}, F^{*t}_1) = promote(D^t_1 \setminus D^t_{lub}, F^{*t}_1)$ by definition of *promote* and Definition 7.4.2.

$promote(D_2^n \setminus D_{lub}^t, F_2^{*t}) \cup promote(D_2^o \setminus D_{lub}^t, F_2^{*t}) = promote(D_2^t \setminus D_{lub}^t, F_2^{*t})$ by definition of *promote* and Definition 7.4.2.
Also,
$F_1^{*o} \cup F_1^{*n} \cup promote(D_1^t \setminus D_{lub}^t, F_1^{*t}) = extend(D_1^t \setminus D_{lub}^t, F_1^{*t})$ via the definitions of *promote* and *extend* and Definition 7.4.2.
$F_2^{*o} \cup F_2^{*n} \cup promote(D_2^t \setminus D_{lub}^t, F_2^{*t}) = extend(D_2^t \setminus D_{lub}^t, F_2^{*t})$ via the definitions of *promote* and *extend* and Definition 7.4.2.
Hence,
$F_{lub}^{*t} = extend(D_1^t \setminus D_{lub}^t, F_1^{*t}) \cup extend(D_2^t \setminus D_{lub}^t, F_2^{*t})$ which corresponds to the definition of F_{lub}^{*t} obtained via Definition 7.2.5. □

When using the abstract interpretation system PLAI of Muthukumar and Hermenegildo [96, 94], the general version of the least upper bound operation is only needed when annotating the program : normally only *one* general version of each predicate is given as output. This implies that for each point in a predicate the system has to compute the upper bound of all abstract constraints generated at that point in the different predicate specialisations. During the analysis phase however (cf. Definition 7.4.7 of procedure-exit), only the $(D_{lub}^n, F_{lub}^{*n})$-component is needed.

7.4.2 Abstract interpretation of a constraint

The abstract interpretation of a constraint C consists of computing the abstraction of C, $\alpha^{\mathcal{DF}}(C)$, and joining it with the current abstract constraint store. Hereby, $\alpha^{\mathcal{DF}}(C)$ itself and the combination of it with the local information are put into the n-component, whereas the combination with information passed on at procedure-entry is put into the o-component of the abstract success constraint.

Definition 7.4.5 (Abstract interpretation of a constraint)
Let $C \in SCons$ and let $((D_c^o, F_c^{*o}), (D_c^n, F_c^{*n}))$ be the abstract call constraint of C. The abstract success constraint $((D_s^o, F_s^{*o}), (D_s^n, F_s^{*n}))$ of C is defined as :

1. $D_s^n = D_s^t \setminus D_c^o$ *where* D_s^t *is obtained by abstract interpretation of* C *on* $Con^{\mathcal{D}}$ *[43, 41] (D_s^t is the set of* all *variables that are definite after abstract interpretation of C);*
2. $D_s^o = D_c^o$;
3. $F_s^{*n} = F_c^{*n} \triangle_{D_s^t} \alpha^{\mathcal{F}}(C)$;
4. $F_s^{*o} = reduce(D_s^t, F_c^{*o}) \cup (reduce(D_s^t, F_c^{*o}) \oplus^{\mathcal{F}} reduce(D_s^t, \alpha^{\mathcal{F}}(C)))$
   $= reduce(D_s^t, F_c^{*o}) \cup (F_c^{*o} \oplus_{D_s^t} \alpha^{\mathcal{F}}(C))$.

It can be shown that $(D_s^t, F_s^{*t}) =^{\mathcal{DF}} (D_c^t, F_c^{*t}) \Lambda^{\mathcal{DF}} \alpha^{\mathcal{DF}}(C)$, based on the definitions of $\Lambda^{\mathcal{DF}}$, $\alpha^{\mathcal{DF}}$ and Definition 7.4.5. Restricting attention to the freeness part of the abstractions, we have that $F_s^{*t} =^{\mathcal{F}} F_c^{*t} \triangle_{D_s^t} \alpha^{\mathcal{F}}(C)$[4]. So the safety of abstract interpretation of a constraint is based on the properties of $\Lambda^{\mathcal{DF}}$ and $\triangle_{D_s^t}$ and the relation of these operations with the freeness operations. We also have that $extend(D_s^t, F_s^{*t}) =^{\mathcal{F}} extend(D_c^t, F_c^{*t}) \Lambda^{\mathcal{F}} \alpha^{\mathcal{F}}(C)$.

[4]One could first compute $\alpha^{\mathcal{DF}}(C) = (D', F'^*)$ and use F'^* instead of $\alpha^{\mathcal{F}}(C)$. However, reduction with D_s^t is still needed afterwards. So it is computationally more efficient to use $\alpha^{\mathcal{F}}(C)$ and then to perform reduction with D_s^t at once.

7.4.3 Procedure-entry

Procedure-entry computes the abstract constraint at the beginning of a clause used to resolve a procedure call, given the abstract call constraint of the call.

Definition 7.4.6 (Procedure-entry)
Let $((D_c^o, F_c^{*o}),(D_c^n, F_c^{*n}))$ be the abstract call constraint of a procedure call $p(Y_1,\ldots,Y_k)$ and let $p(Z_1,\ldots,Z_k)$ be the head of a clause[5] used to resolve the call. Then, the abstract constraint at the beginning of the clause is $AC_{in} = AC_{entry}\rho$ with $AC_{entry} = ((D^o_{entry}, F^o_{entry}), (\emptyset,\emptyset))$ and $(D^o_{entry}, F^o_{entry}) = \exists^{\mathcal{DF}}_{\{Y_1,\ldots,Y_k\}}(D_c^t, F_c^{*t})$ and $\rho = \{Y_1 \leftarrow Z_1,\ldots,Y_k \leftarrow Z_k\}$ (renaming).

It follows immediately from this definition that $(D^t_{in}, F^{*t}_{in}) = (\exists^{\mathcal{DF}}_{\{Y_1,\ldots,Y_k\}}(D_c^t, F_c^{*t}))\rho$.

7.4.4 Procedure-exit

Procedure-exit computes the abstract success constraint of a procedure call, given its abstract call constraint and the abstract constraints at the end of each clause used to resolve the call.

Definition 7.4.7 (Procedure-exit)
Let $((D_c^o, F_c^{*o}),(D_c^n, F_c^{*n}))$ be the abstract call constraint of a call $p(Y_1,\ldots,Y_k)$. Let the head of each clause Cl_i defining p/k be of the form $p(Z_1,\ldots,Z_k)$ $(1 \le i \le m)$.
Let $((D_i^o, F_i^{*o}),(D_i^n, F_i^{*n}))$ be the abstract constraint at the end of clause Cl_i and let $(D^n_{exit}, F^{*n}_{exit}) = lub^{\mathcal{DF}}(\{\ldots,(\exists^{\mathcal{DF}}_{\{Z_1,\ldots,Z_k\}}(D_i^n, F_i^{*n}))\rho^{-1},\ldots\})$ with $\rho^{-1} = \{Z_1 \leftarrow Y_1,\ldots,Z_k \leftarrow Y_k\}$ (renaming). The abstract success constraint $((D_s^o, F_s^{*o}),(D_s^n, F_s^{*n}))$ of the call is defined as :

1. *$D_s^n = D_s^t \setminus D_c^o$ where D_s^t is obtained by performing procedure-exit on $Con^{\mathcal{D}}$ [43, 41] (D_s^t is the set of all variables that are definite after interpretation of the procedure call);*
2. *$D_s^o = D_c^o$;*
3. *$F_s^{*n} = F_c^{*n} \triangle_{D_s^t} F^{*n}_{exit}$;*
4. *$F_s^{*o} = reduce(D_s^t, F_c^{*o}) \cup (reduce(D_s^t, F_c^{*o}) \oplus^{\mathcal{F}} reduce(D_s^t, F^{*n}_{exit}))$*
   $= reduce(D_s^t, F_c^{*o}) \cup (F_c^{*o} \oplus_{D_s^t} F^{*n}_{exit})$.

As in the case of the freeness and minimal freeness abstraction, part of the information is recomputed at procedure-exit.

It can be shown that $(D_s^t, F_s^{*t}) =^{\mathcal{DF}} (D_c^t, F_c^{*t})\ \Lambda^{\mathcal{DF}} (D^n_{exit}, F^{*n}_{exit})$, based on the definitions of $\Lambda^{\mathcal{DF}}$ and Definition 7.4.7. Restricting attention to the freeness part of the abstractions, we have that $F_s^{*t} =^{\mathcal{F}} F_c^{*t} \triangle_{D_s^t} F^{*n}_{exit}$. So the safety of procedure-exit is based on the properties of $\Lambda^{\mathcal{DF}}$ and $\triangle_{D_s^t}$ and the relation of these operations with the freeness operations. We also have that $extend(D_s^t, F_s^{*t}) =^{\mathcal{F}} extend(D_c^t, F_c^{*t})\ \Lambda^{\mathcal{F}}\ extend(D^n_{exit}, F^{*n}_{exit})$.

7.4.5 Efficiency considerations

The efficiency considerations are similar to those of the freeness and minimal freeness abstraction.

[5] The clause is assumed to be renamed apart from the variables in the abstract call constraint.

7.4.5.1 Time efficiency

To avoid the recomputation at procedure-exit, an alternative definition of a compound abstract constraint (and the associated abstract operations) can be formulated. The idea is the same as in the case of the freeness abstraction (Section 5.4.5.1) : in the alternative definition, the combination of information passed down at procedure-entry with local information gathered during the procedure execution is added to the n-component of a compound abstract constraint rather than to its o-component. This time-efficient approach may cause some loss of precision.

7.4.5.2 Space efficiency

Compound abstract constraints (both with the original and with the time-efficient approach) can be reduced by keeping the F^{*o} and F^{*n} components disjoint. It means that, at each step in the analysis, the old component is adjusted by removing those sets of variables that also occur within the new component; the new component itself is not affected. For example, the definition of abstract interpretation of a constraint becomes :

Definition 7.4.8 (Abstract interpretation of a constraint (space-optimised)) *Let $C \in SCons$ and let $((D_c^o, F_c^{*o}), (D_c^n, F_c^{*n}))$ be the abstract call constraint of C. The abstract success constraint $((D_s^o, F_s^{*o}), (D_s^n, F_s^{*n}))$ of C is defined as :*

1. $D_s^n = D_s^t \setminus D_c^o$ *where* D_s^t *is obtained by abstract interpretation of C on $Con^{\mathcal{D}}$ [43, 41] (D_s^t is the set of* all *variables that are definite after abstract interpretation of C);*
2. $D_s^o = D_c^o$.
3. $F_s^{*n} = F_c^{*n} \triangle_{D_s^t} \alpha^{\mathcal{F}}(C)$;
4. $F_s^{*o} = (\ reduce(D_s^t, F_c^{*o}) \cup (F_c^{*o} \oplus_{D_s^t} \alpha^{\mathcal{F}}(C))\ ) \setminus F_s^{*n}$.

The definition of the other abstract operations is adjusted in a similar way. The order relation must be redefined as follows : $((D_1^o, F_1^{*o}), (D_1^n, F_1^{*n})) \leq^{\mathcal{DF}} ((D_2^o, F_2^{*o}), (D_2^n, F_2^{*n}))$ iff $D_1^o \supseteq D_2^o, D_1^n \supseteq D_2^n$, $extend(D_1^o \setminus D_2^o, F_1^{*o}) \subseteq F_2^{*t}$ *and* $extend(D_1^n \setminus D_2^n, F_1^{*n}) \subseteq F_2^{*n}$.

It can be shown that no information is lost by performing the optimisation (cf. Section 5.4.5.2).

The reduction optimises space but the effect on time efficiency is not clear. The analysis time may improve due to the computation with smaller abstract constraints. However, keeping F^{*o} and F^{*n} disjoint incurs a time overhead and potentially implies a decrease of time efficiency.

7.5 Examples

We illustrate the $\mathcal{DF}$ analysis on the same program examples as used for the freeness and minimal freeness analysis.

First of all, consider the following simple program involving only numerical constraints. We assume that the program is called with all variables being free (i.e. starting with the empty constraint store).

?- $(AC_0)\ Z = 1,\ (AC_1)\ p(X, Y, Z, T)\ (AC_2)$.

$p(X, Y, Z, T) \leftarrow (AC_3)\ X = Y + Z\ (AC_4)$.
$p(X, Y, Z, T) \leftarrow (AC_5)\ X = Y + T\ (AC_6)$.

The compound $\mathcal{DF}$ abstract constraints (not space-optimised) derived at each program point are the following :

$AC_0 = ((\emptyset, \emptyset),\ (\emptyset, \emptyset))$
$AC_1 = ((\emptyset, \emptyset),\ (\{Z\}, \emptyset))$
$AC_2 = ((\emptyset, \emptyset),\ (\{Z\}, \{\{X, Y\}, \{X, Y, T\}\}))$

$AC_3 = ((\{Z\}, \emptyset),\ (\emptyset, \emptyset))$
$AC_4 = ((\{Z\}, \emptyset),\ (\emptyset, \{\{X, Y\}\}))$

$AC_5 = ((\{Z\}, \emptyset),\ (\emptyset, \emptyset))$
$AC_6 = ((\{Z\}, \emptyset),\ (\emptyset, \{\{X, Y, T\}\}))$

Space-optimisation does not yield further improvements for this example. All AC_i with $i \geq 1$ indicate that Z is definite. The $\mathcal{DF}$ abstract constraints AC_2, AC_4 and AC_6 are more compact than the corresponding abstract constraints obtained with the freeness analysis. Compared with the $\mathcal{M}$ analysis, AC_4 and AC_6 are more compact here. Moreover, even if abstract constraints contain the same number of sets in the $\mathcal{M}$ and $\mathcal{DF}$ analysis, the definite variables are extracted from the freeness part and hence are no longer involved in the abstract operations on that part.

The second example involves both numerical and unification constraints. Again, we assume that the program starts off with the empty constraint store.

?- $(AC_0)\ Z = 1,\ (AC_1)\ p(X, Y, Z, T)\ (AC_2)$.

$p(X, Y, Z, T) \leftarrow (AC_3)\ X = f(Z)\ (AC_4)\ ,\ Z + T = 0\ (AC_5)$.
$p(X, Y, Z, T) \leftarrow (AC_6)\ X = g(Y)\ (AC_7)\ ,\ Y - T = 0\ (AC_8)$.

The compound abstract constraints derived at each program point are :

$AC_0 = ((\emptyset, \emptyset),\ (\emptyset, \emptyset))$
$AC_1 = ((\emptyset, \emptyset),\ (\{Z\}, \emptyset))$
$AC_2 = ((\emptyset, \emptyset),\ (\{Z\}, \{\{T\}, \{X\}, \{X, Y\}, \{Y, T\}, \{X, T\}, \{X, Y, T\}\}))$

$AC_3 = ((\{\{Z\}\}, \emptyset),\ (\emptyset, \emptyset))$
$AC_4 = ((\{\{Z\}\}, \emptyset),\ (\{X\}, \emptyset))$
$AC_5 = ((\{\{Z\}\}, \emptyset),\ (\{X, T\}, \emptyset))$

$AC_6 = ((\{\{Z\}\}, \emptyset),\ (\emptyset, \emptyset))$
$AC_7 = ((\{\{Z\}\}, \emptyset),\ (\emptyset, \{\{X\}, \{X, Y\}\}))$
$AC_8 = ((\{\{Z\}\}, \emptyset),\ (\emptyset, \{\{X\}, \{X, Y\}, \{Y, T\}, \{X, T\}, \{X, Y, T\}\}))$

Space-optimisation does not yield any further improvements. Note that the $\mathcal{DF}$ abstractions are in general much more compact than the corresponding freeness abstractions. Also compared with the $\mathcal{M}$ analysis, AC_4, AC_5, AC_7 and AC_8 are now (much) more compact.

Note that AC_2 is larger than in the $\mathcal{M}$ case due to the fact that no extra definiteness information (besides Z) is available and the freeness part is not minimised here.

AC_5 indicates that at the end of the first clause X, Z and T are definite. At the end of the second clause (cf. AC_8), only Z is definite, X is possibly non-free and Y and T are free. So, at AC_2 where the results of the two clauses are put together, Z is definite, X and T are possibly non-free (T is possibly non-free since it is definite in one clause but free in the other clause) and Y is free.

The last example is the well-known sumlist program, which again involves mixed constraints. It defines the relation between a list L and the sum S of the list elements. We assume that the program is called with the first argument being definite and the second argument free.

?- (AC_0) $sumlist(List, Sum)$ (AC_1).

$sumlist(L, S) \leftarrow$
$\quad (AC_2)\ L = [\,]$,
$\quad (AC_3)\ S = 0$ (AC_4).
$sumlist(L, S) \leftarrow$
$\quad (AC_5)\ L = [H \mid T]$,
$\quad (AC_6)\ S = H + S1$,,
$\quad (AC_7)$ $sumlist(T, S1)$ (AC_8).

The $\mathcal{DF}$ abstract constraints obtained at each program point are shown below. For simplicity, we only present the $AC_i^t = (D_i^t, F_i^{*t})$ instead of showing the components AC_i^o and AC_i^n. Recall that the AC_i^t are sufficient to derive mode and dependency information.

$AC_0^t = (\{List\}, \emptyset)$
$AC_1^t = (\{List, Sum\}, \emptyset)$

$AC_2^t = (\{L\}, \emptyset)$
$AC_3^t = (\{L\}, \emptyset)$
$AC_4^t = (\{L, S\}, \emptyset)$

$AC_5^t = (\{L\}, \emptyset)$
$AC_6^t = (\{L, H, T\}, \emptyset)$
$AC_7^t = (\{L, H, T\}, \{\{S, S1\}\})$
$AC_8^t = (\{L, H, T, S, S1\}, \emptyset)$

The results show that, after executing $sumlist/2$ with the given call pattern, the second argument is definite. In the second clause, the definiteness of L in AC_5^t is propagated onto H and T via $L = [H \mid T]$ (cf. the presence of $\{H\}$ and $\{T\}$ in the D-component of AC_6^t). Just before the recursive call (in AC_7^t) S and $S1$ are still free but there is a (entailed) possible dependency between them.
The $\mathcal{DF}$ abstract constraints are much more compact than the abstract constraints obtained with the freeness analysis. Compared with the $\mathcal{M}$ analysis, mostly definiteness information is inferred now such that the freeness part is reduced to the empty set (except for AC_7^t).

Chapter 8

Minimal freeness exploiting definiteness

In this chapter we present an improved freeness abstraction, called the $\mathcal{DM}$ abstraction, which is based on the minimal freeness abstraction and which uses additional knowledge about definiteness of program variables. It combines the two approaches to compress the freeness abstraction that were presented in Chapters 6 and 7 and results in an efficient and practical analysis.

The first section of this chapter introduces the $\mathcal{DM}$ abstraction. The second section describes the concrete and abstract domain, together with the concretisation and abstraction function. The primitive and higher-level abstract operations are defined in sections three and four. Finally, the $\mathcal{DM}$ analysis is illustrated on some program examples.

8.1 Introduction

In Chapters 6 and 7 we presented two approaches to reduce the size of the freeness abstract constraints.

In the minimal freeness abstraction (Chapter 6), the definite variables occur as possibly non-free variables, i.e. as a set of singletons each containing a definite variable. Although the presence of definite variables is reduced to the set of singletons, the abstract operations still have to take these singletons into account, for example when computing the closure under union. Efficiency of the analysis can therefore be improved by separating out the definite variables, as they play a very specific role in the propagation of possible non-freeness.

In the $\mathcal{DF}$ abstraction (Chapter 7), the definite variables and their related dependencies are separated out of the original freeness abstraction. Although this considerably reduces the size of the abstract constraints, a further compression is possible by minimising the remaining set of dependencies between non-definite variables.

As a consequence, the combination of the approaches in Chapters 6 and 7 (which is possible as they are orthogonal) results in a practical and efficient mode analysis.

8.2 Concrete and abstract domain

The concrete domain for the $\mathcal{DM}$ abstraction is the same as for the freeness abstraction, i.e. $Con^{\mathcal{C}} = \wp(Cons)$.

Definition 8.2.1 (Abstract domain)
The abstract domain is $Con^{\mathcal{DM}} = \{(D, M^*) | D \in \wp(Var), M^* \in \wp(\wp_\emptyset(Var \backslash D)), min(M^*) = M^*\} \cup \{\bot\}$ *with min as defined in Definition 6.2.3.*

The abstract constraint $\bot$ denotes definite unsatisfiability or unreachability.

Given a $CS \in Con^{\mathcal{C}}$, the abstraction $\alpha^{\mathcal{DM}}(CS)$ can be obtained in two ways. A first approach is to start from the original freeness abstraction and split off the definite variables and related dependencies. Afterwards the remaining set of dependencies between non-definite variables is minimised. The latter operation may cause some loss of precision. An equivalent but somewhat more efficient approach is to compute the minimal freeness abstraction (possibly losing some precision) and then split off the definite part (at that point minimised to a set of singletons of definite variables). The latter approach is more efficient than the first one, as the minimal freeness abstraction can be obtained without first computing the full freeness abstraction (cf. Section 6.2). So $\alpha^{\mathcal{DM}}(CS)$ is defined in terms of $\alpha^{\mathcal{M}}(CS)$ and (a subset of) the set of definite variables in CS.

Definition 8.2.2 (Abstraction function)
Let $CS \in Con^{\mathcal{C}}$. *Then* $\alpha^{\mathcal{DM}}(\emptyset) = \bot$, *otherwise* $\alpha^{\mathcal{DM}}(CS) = (D, M^*)$ *where* D *could be given by* $defvars(\alpha^{\mathcal{D}}(CS))$ *and* $M^* = compl(D, \alpha^{\mathcal{M}}(CS))$ *(recall : $defvars(\alpha^{\mathcal{D}}(CS))$ denotes the set of definite variables extracted from $\alpha^{\mathcal{D}}(CS)$ [43, 41] and $compl(D, SS) = \{S \in SS \mid S \cap D = \emptyset\}$).*

Figure 8.1 illustrates the relation between the different abstractions of a $CS \in Con^{\mathcal{C}}$; $F = \alpha^{\mathcal{F}}(CS)$, $M = \alpha^{\mathcal{M}}(CS)$, $(D, F^*) = \alpha^{\mathcal{DF}}(CS)$ and $(D, M^*) = \alpha^{\mathcal{DM}}(CS)$. The operation $extend^m$ transforms a $Con^{\mathcal{DM}}$ to a $Con^{\mathcal{M}}$ abstraction.

Definition 8.2.3 (extend^m)
Let $(D, M^*) \in Con^{\mathcal{DM}}$. *Then* $extend^m(D, M^*) = M^* \cup \{\{X\} \mid X \in D\}$.

Example 8.2.1
Let $CS \equiv \{(X + Y + Z = 3 \wedge Y + Z = 2 \wedge T = f(Z)), (Y + Z = 5 \wedge T = a \wedge X = 3)$. *The set of definite variables in CS is* $\{X\}$ *as the first constraint in CS entails* $X = 1$ *and the second constraint contains* $X = 3$. *Then*

$$\alpha^{\mathcal{F}}(CS) = close(\{\{X\}, \{Y, Z\}, \{T\}, \{T, Z\}, \{Y, T\}\}),$$
$$\alpha^{\mathcal{DF}}(CS) = (\{X\}, \{\{Y, Z\}, \{T\}, \{T, Z\}, \{Y, T\}, \{Y, Z, T\}\}),$$
$$\alpha^{\mathcal{M}}(CS) = \{\{X\}, \{Y, Z\}, \{T\}, \{T, Z\}, \{Y, T\}\} \text{ and}$$
$$\alpha^{\mathcal{DM}}(CS) = (\{X\}, \{\{Y, Z\}, \{T\}, \{T, Z\}, \{Y, T\}\}).$$

To simplify the presentation, we do not explicitly deal with $\bot$ in the definition of the operations on $Con^{\mathcal{DM}}$. Extension of the operations is straightforward, having in mind that $\bot$ is the minimal element in $Con^{\mathcal{DM}}$.

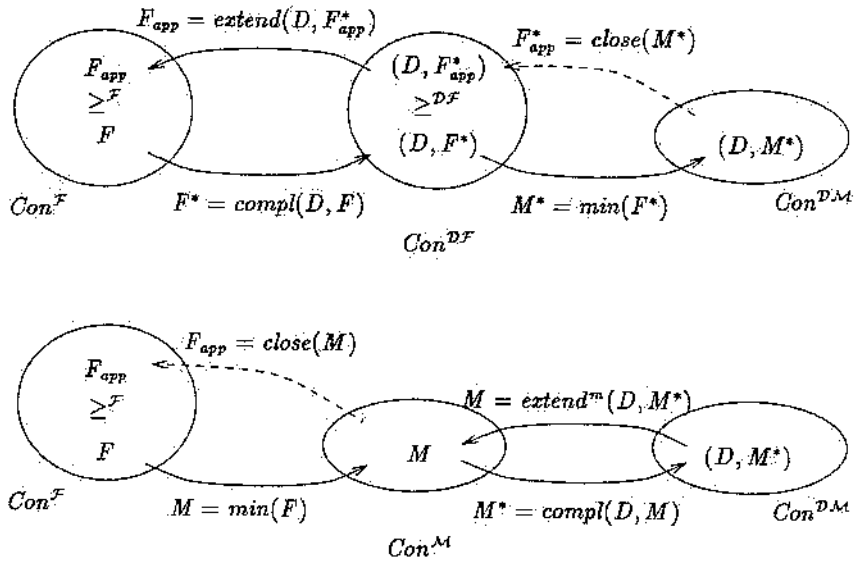


Figure 8.1: Relation between the $Con^{\mathcal{F}}$, $Con^{\mathcal{M}}$, $Con^{\mathcal{DF}}$ and $Con^{\mathcal{DM}}$ abstraction

Definition 8.2.4 (Abstract computational order)
Let $(D_1, M_1^*), (D_2, M_2^*) \in Con^{\mathcal{DM}}$.
Then $(D_1, M_1^*) \leq^{\mathcal{DM}} (D_2, M_2^*)$ *iff* $D_1 \supseteq D_2$ *and* $extend^m(D_1 \setminus D_2, M_1^*) \subseteq M_2^*$.

The minimal element of $Con^{\mathcal{DM}}$ is $\perp$, the maximal element is $(\emptyset, \{\{X\} \mid X \in Var\})$.
The equality relation $=^{\mathcal{DM}}$ induced by $\leq^{\mathcal{DM}}$ is the following : $(D_1, M_1^*) =^{\mathcal{DM}} (D_2, M_2^*)$ iff $D_1 = D_2$ and $M_1^* = M_2^*$.

Definition 8.2.5 (Abstract least upper bound)
Let $(D_1, M_1^*), (D_2, M_2^*) \in Con^{\mathcal{DM}}$. *Then* $lub^{\mathcal{DM}}((D_1, M_1^*), (D_2, M_2^*)) = (D, M^*)$ *with*
$D = D_1 \cap D_2$ *and* $M^* = min(extend^m(D_1 \setminus (D_1 \cap D_2), M_1^*) \cup extend^m(D_2 \setminus (D_1 \cap D_2), M_2^*))$
$= min(M_1^* \cup M_2^* \cup \{\{X\} \mid X \in (D_1 \cup D_2) \setminus (D_1 \cap D_2)\})$.

This operation can easily be generalised to compute the least upper bound of n ($n > 2$) abstract constraints. In the sequel we assume that $lub^{\mathcal{DM}}$ applies to a set of abstract constraints.
An abstract constraint of $Con^{\mathcal{DM}}$ subsumes mode and dependency information as follows.

Proposition 8.2.1
Let $(D, M^*) \in Con^{\mathcal{DM}}$.
Then the following holds for each $CS \in Con^{\mathcal{C}}$ *such that* $\alpha^{\mathcal{DM}}(CS) \leq^{\mathcal{DM}} (D, M^*)$:

1. *If* $X \in D$, *then* X *is a definite variable in* CS *(mode* **d***), else if* $\{X\} \notin M^*$ *then* X *is free in* CS *(mode* **f***). Otherwise nothing can be said about* X *(mode* **a***).*

2. *If* $\{X_1, \ldots, X_n\} \in M^*$ *or there exist* $S_1, \ldots, S_m \in M^*$ $(m \geq 2)$ *such that* $S_1 \cup \ldots \cup S_m = \{X_1, \ldots, X_n\}$, *then* CS possibly *establishes a dependency between the variables* $X_1, \ldots, X_n$, *i.e. (further) constraining all but one of the variables* may *(further) constrain the remaining variable. Otherwise, there is certainly no dependency* $\{X_1, \ldots, X_n\}$ *in each* $C \in CS$.

Finally, the concretisation function $\gamma^{\mathcal{DM}}$ is determined by $\alpha^{\mathcal{DM}}$ and $\leq^{\mathcal{DM}}$.

Definition 8.2.6 (Concretisation function)
Let $AC \in Con^{\mathcal{DM}}$. *Then* $\gamma^{\mathcal{DM}}(AC) = \bigcup \{CS \in Con^{\mathcal{C}} \mid \alpha^{\mathcal{DM}}(CS) \leq^{\mathcal{DM}} AC\}$.

Proposition 8.2.2 describes the relation between the concrete and abstract domain.

Proposition 8.2.2
$(Con^{\mathcal{C}}, \leq^{\mathcal{C}}) \leftrightarrows (Con^{\mathcal{DM}}, \leq^{\mathcal{DM}})$ *is a Galois insertion.*

PROOF Similar to the proof of Proposition 5.2.4. □

8.3 Primitive abstract operations

The primitive abstract operations are abstract conjunction and abstract projection. The operations are defined in terms of the operations on the D and M^* parts of the abstract constraints. For simplicity, we do not explicitly deal with $\bot$.

We first define some auxiliary operations.

Definition 8.3.1 (reduce^m)
Let $D \in \wp(Var)$ *and* $M^* \in \wp(\wp_\emptyset(Var))$.
Then $reduce^m(D, M^*) = min(\{S \setminus D \mid S \in M^*\}) \setminus \{\emptyset\}$.

Definition 8.3.2 ($\oplus^m$ w.r.t. D)
Let $D \in \wp(Var)$ *and* $M_1^*, M_2^* \in \wp(\wp_\emptyset(Var))$. *Then* $M_1^* \oplus_D^m M_2^*$ *is defined as*

$$M_1^* \oplus_D^m M_2^* = reduce^m(D, M_1^*) \oplus^{\mathcal{M}} reduce^m(D, M_2^*).$$

Definition 8.3.3 (Minimal abstract conjunction w.r.t. D)
Let $D \in \wp(Var)$ *and* $M_1^*, M_2^* \in \wp(\wp_\emptyset(Var))$. *The minimal abstract conjunction of* M_1^* *and* M_2^* *with respect to* D, *denoted* $M_1^* \triangle_D^m M_2^*$, *is defined as*

$$M_1^* \triangle_D^m M_2^* = reduce^m(D, M_1^*) \wedge^{\mathcal{M}} reduce^m(D, M_2^*).$$

Definition 8.3.4 (Abstract conjunction)
Let $(D_1, M_1^*), (D_2, M_2^*) \in Con^{\mathcal{DF}}$. *Then* $(D_1, M_1^*) \wedge^{\mathcal{DM}} (D_2, M_2^*) = (D, M^*)$ *where* D *is obtained using the abstract conjunction on* $Con^{\mathcal{D}}$ *[43, 41] and* $M^* = M_1^* \triangle_D^m M_2^*$.

D satisfies $D \supseteq D_1 \cup D_2$ (D_1 and D_2 are only parts of the $Con^{\mathcal{D}}$ abstraction, which also contains definite dependencies; so D may be strictly larger than $D_1 \cup D_2$ due to definiteness propagation via these dependencies).

The definition of M^* could be replaced by the following equivalent and straightforward definition : $M^* = compl(D, extend^m(D_1, M_1^*) \wedge^{\mathcal{M}} extend^m(D_2, M_2^*))$. However, computing M^* via $M_1^* \ \triangle_D^m \ M_2^*$ is more efficient, since the conjunction is performed on smaller abstract constraints. The idea is to join the definiteness parts first and then to propagate the obtained definiteness information (via $reduce^m$) onto the freeness parts M_1^* and M_2^* before joining them. More precisely, this means removing the singletons that contain a definite variable of D from M_1^* and M_2^*, such that these definite variables are not taken into account when performing conjunction. The proof of the equivalence between the two definitions is similar to the proof of Proposition 7.3.1 (now also taking into account min).

The safety of abstract conjunction is based on the above equivalence and the safety of $\wedge^{\mathcal{M}}$. Safety conditions can also be formulated on the separate D and M^* components of the $\mathcal{DM}$ abstractions.

Proposition 8.3.1 (Safety of abstract conjunction)
Let $C_1, C_2 \in SCons$ and $(D_1, M_1^*), (D_2, M_2^*), (D, M^*) \in Con^{\mathcal{DM}}$ with $(D_1, M_1^*) \wedge^{\mathcal{DM}} (D_2, M_2^*) = (D, M^*)$. Then

1. *if $defvars(\alpha^{\mathcal{D}}(C_1)) \leq^{\mathcal{D}} D_1$ and $defvars(\alpha^{\mathcal{D}}(C_2)) \leq^{\mathcal{D}} D_2$, then $defvars(\alpha^{\mathcal{D}}(C_1 \wedge C_2)) \leq^{\mathcal{D}} D$.*
2. *if $\alpha^{\mathcal{M}}(C_1) \leq^{\mathcal{M}} extend^m(D_1, M_1^*)$ and $\alpha^{\mathcal{M}}(C_2) \leq^{\mathcal{M}} extend^m(D_2, M_2^*)$, then $\alpha^{\mathcal{M}}(C_1 \wedge C_2) \leq^{\mathcal{M}} extend^m(D, M^*)$.*
3. *if $\alpha^{\mathcal{DM}}(C_1) \leq^{\mathcal{DM}} (D_1, M_1^*)$ and $\alpha^{\mathcal{DM}}(C_2) \leq^{\mathcal{DM}} (D_2, M_2^*)$, then $\alpha^{\mathcal{DM}}(C_1 \wedge C_2) \leq^{\mathcal{DM}} (D, M^*)$.*

PROOF Similar to the proof of Proposition 7.3.2. □

The abstract projection operation is defined as in Definition 7.3.5.

Definition 8.3.5 (Abstract projection)
Let $(D, M^*) \in Con^{\mathcal{DM}}$ and $V \subseteq Var$. Then $\exists_V^{\mathcal{DM}}(D, M^*) = (D_p, M_p^*)$ with $D_p = D \cap V$ and $M_p^* = \{S \in M^* \mid S \subseteq V\}$.

The formulation and proof of its safety corresponds to Proposition 7.3.4 (with $\mathcal{DF}$ replaced by $\mathcal{DM}$ and $extend$ by $extend^m$).

8.4 Abstract operations

For simplicity, we do not explicitly consider $\bot$ in the definitions of the abstract operations. The safety of the abstract operations depends on the safety of abstract conjunction and abstract projection and on the relation of the operations with those defined in previous chapters.

8.4.1 Compound abstract constraints

To obtain precise results at procedure-exit, an abstract constraint AC is split up in two components, AC^o and AC^n. It is then called a *compound* abstract constraint.

Definition 8.4.1 (Compound abstract constraint)
A compound abstract constraint AC is either $\bot$ or $(AC^o, AC^n) = ((D^o, M^{*o}), (D^n, M^{*n}))$ with $D^o, D^n \in \wp(Var)$ and $M^{*o}, M^{*n} \in \wp(\wp_\emptyset(Var \setminus D))$ such that $min(M^{*o}) = M^{*o}$ and $min(M^{*n}) = M^{*n}$. The component AC^o contains the information passed on at procedure-entry and the combination of it with the information gathered during local analysis of the procedure body; AC^n contains the information that is obtained from the constraints gathered during local analysis of the procedure body.

Property 8.4.1
For each $((D^o, M^{*o}), (D^n, M^{*n}))$ holds that
1. *$compl(D^o \cup D^n, M^{*o}) = M^{*o}$ (i.e. M^{*o} contains no $(D^o \cup D^n)$-variables)*
2. *$compl(D^o \cup D^n, M^{*n}) = M^{*n}$ (i.e. M^{*n} contains no $(D^o \cup D^n)$-variables)*
3. *$D^o \cap D^n = \emptyset$*
4. *M^{*o} and M^{*n} are not necessarily disjoint.*

The abstract domain then consists of compound abstract constraints rather than of abstract constraints as such.

Definition 8.4.2 (AC^t)
Let $(AC^o, AC^n) = ((D^o, M^{*o}), (D^n, M^{*n}))$ be a compound abstract constraint. Then $AC^t = (D^t, M^{*t}) = (D^o \cup D^n, min(M^{*o} \cup M^{*n}))$.

Mode and dependency information is inferred from AC^t (cf. Proposition 8.2.1).

Definition 8.4.3 (Order relation)
Let AC_1 and AC_2 be two compound abstract constraints. Then[1],

$$((D_1^o, M_1^{*o}), (D_1^n, M_1^{*n})) \leq^{\mathcal{DM}} ((D_2^o, M_2^{*o}), (D_2^n, M_2^{*n})) \text{ iff}$$
$$D_1^o \supseteq D_2^o, D_1^n \supseteq D_2^n, extend^m(D_1^o \setminus D_2^o, M_1^{*o}) \subseteq close(M_2^{*o}) \text{ and}$$
$$extend^m(D_1^n \setminus D_2^n, M_1^{*n}) \subseteq close(M_2^{*n}).$$

This order induces the following equality relation :
$((D_1^o, M_1^{*o}), (D_1^n, M_1^{*n})) =^{\mathcal{DM}} ((D_2^o, M_2^{*o}), (D_2^n, M_2^{*n}))$ iff $D_1^o = D_2^o$, $D_1^n = D_2^n$, $M_1^{*o} = M_2^{*o}$ and $M_1^{*n} = M_2^{*n}$.

Definition 8.4.4 (Least upper bound)
$lub^{\mathcal{DM}}(((D_1^o, M_1^{*o}), (D_1^n, M_1^{*n})), ((D_2^o, M_2^{*o}), (D_2^n, M_2^{*n}))) = ((D_{lub}^o, M_{lub}^{*o}), (D_{lub}^n, M_{lub}^{*n}))$ with
1. *$D_{lub}^o = (D_1^o \cap D_2^o) \cup (D_1^o \cap D_2^n) \cup (D_1^n \cap D_2^o)$* [2]
2. *$D_{lub}^n = (D_1^n \cap D_2^n)$*

[1] The symbol $\leq^{\mathcal{DM}}$ is overloaded. Depending on the context, it denotes the order between compound abstract constraints or between plain abstract constraints (Definition 8.2.4).

[2] D_{lub}^o can be computed more efficiently from D_{lub}^t supplied by the definiteness analysis and D_{lub}^n : $D_{lub}^o = D_{lub}^t \setminus D_{lub}^n$.

3. $M^{*o}_{lub} = min(M^{*o}_1 \cup M^{*o}_2 \cup promote^m(D^o_1 \setminus D^t_{lub}, M^{*t}_1) \cup promote^m(D^o_2 \setminus D^t_{lub}, M^{*t}_2)$
$\cup\ promotecombs^m(D^n_1 \setminus D^t_{lub}, M^{*o}_1) \cup promotecombs^m(D^n_2 \setminus D^t_{lub}, M^{*o}_2))$
(after simplification, avoiding the generation of non-minimal sets where possible:)
$= min(M^{*o}_1 \cup M^{*o}_2 \cup \{\{X\} \mid X \in (D^o_1 \cup D^o_2) \setminus D^t_{lub}\}$
$\cup\ promotecombs^m(D^n_1 \setminus D^t_{lub}, M^{*o}_1) \cup promotecombs^m(D^n_2 \setminus D^t_{lub}, M^{*o}_2)$
$\cup\ promotecombs^m(D^o_1 \setminus D^t_{lub}, M^{*n}_1) \cup promotecombs^m(D^o_2 \setminus D^t_{lub}, M^{*n}_2))$

4. $M^{*n}_{lub} = min(M^{*n}_1 \cup M^{*n}_2 \cup promote^m(D^n_1 \setminus D^t_{lub}, M^{*n}_1) \cup promote^m(D^n_2 \setminus D^t_{lub}, M^{*n}_2))$
$= min(extend^m(D^n_1 \setminus D^t_{lub}, M^{*n}_1) \cup extend^m(D^n_2 \setminus D^t_{lub}, M^{*n}_2))$
$= min(M^{*n}_1 \cup M^{*n}_2 \cup \{\{X\} \mid X \in (D^n_1 \cup D^n_2) \setminus D^t_{lub}\})$

where $promote^m(D, M^*) = \{\{X\} \mid X \in D\} \cup \{\{X\} \cup S_2 \mid X \in D, S_2 \in M^*\}$ *and*
$promotecombs^m(D, M^*) = \{\{X\} \cup S_2 \mid X \in D, S_2 \in M^*\}$.

The following proposition shows that this definition of least upper bound is compatible with Definition 8.2.5 of the least upper bound on plain abstract constraints.

Proposition 8.4.1
Let $((D^o_1, M^{*o}_1), (D^n_1, M^{*n}_1))$, $((D^o_2, M^{*o}_2), (D^n_2, M^{*n}_2))$ *be two compound abstract constraints and* $((D^o_{lub}, M^{*o}_{lub}), (D^n_{lub}, M^{*n}_{lub})) = lub^{\mathcal{DM}}(((D^o_1, M^{*o}_1), (D^n_1, M^{*n}_1)), ((D^o_2, M^{*o}_2), (D^n_2, M^{*n}_2)))$.
Then $lub^{\mathcal{DM}}((D^t_1, M^{*t}_1), (D^t_2, M^{*t}_2)) = (D^t_{lub}, M^{*t}_{lub})$ *where the latter* $lub^{\mathcal{DM}}$ *is the one of Definition 8.2.5.*

PROOF Similar to the proof of Proposition 7.4.1. □

When using the abstract interpretation system PLAI of Muthukumar and Hermenegildo [96, 94], the general version of the least upper bound operation is only needed when annotating the program : normally only *one* general version of each predicate is given as output. This implies that for each point in a predicate the system has to compute the upper bound of all abstract constraints generated at that point in the different predicate specialisations. During the analysis phase however (cf. Definition 8.4.7 of procedure-exit), only the $(D^n_{lub}, M^{*n}_{lub})$-component is needed.

8.4.2 Abstract interpretation of a constraint

The abstract interpretation of a constraint C consists of computing the abstraction of C, $\alpha^{\mathcal{DM}}(C)$, and joining it with the current abstract constraint store. Hereby, $\alpha^{\mathcal{DM}}(C)$ itself and the combination of it with the local information are put into the n-component, whereas the combination with information passed on at procedure-entry is put into the o-component of the abstract success constraint.

Definition 8.4.5 (Abstract interpretation of a constraint)
Let $C \in SCons$ *and let* $((D^o_c, M^{*o}_c), (D^n_c, M^{*n}_c))$ *be the abstract call constraint of* C. *The abstract success constraint* $((D^o_s, M^{*o}_s), (D^n_s, M^{*n}_s))$ *of* C *is defined as :*

1. $D^n_s = D^t_s \setminus D^o_c$ *where* D^t_s *is obtained by abstract interpretation of* C *on* $Con^{\mathcal{D}}$ *[43, 41]* *(D^t_s is the set of* all *variables that are definite after abstract interpretation of* C*);*
2. $D^o_s = D^o_c$.
3. $M^{*n}_s = M^{*n}_c \triangle^m_{D^t_s} \alpha^{\mathcal{M}}(C)$;

4. $M_s^{*o} = reduce^m(D_s^t, M_c^{*o}) \cup (reduce^m(D_s^t, M_c^{*o}) \oplus^{\mathcal{M}} reduce^m(D_s^t, \alpha^{\mathcal{M}}(C)))$
$= reduce^m(D_s^t, M_c^{*o}) \cup (M_c^{*o} \oplus^m_{D_s^t} \alpha^{\mathcal{M}}(C))$.

It can be shown that $(D_s^t, M_s^{*t}) =^{\mathcal{DM}} (D_c^t, M_c^{*t}) \Lambda^{\mathcal{DM}} \alpha^{\mathcal{DM}}(C)$, based on the definitions of $\Lambda^{\mathcal{DM}}$, $\alpha^{\mathcal{DM}}$ and Definition 8.4.5. Restricting attention to the freeness part of the abstractions, we have that $M_s^{*t} =^{\mathcal{M}} M_c^{*t} \ \triangle^m_{D_s^t} \alpha^{\mathcal{M}}(C)$[3]. So the safety of abstract interpretation of a constraint is based on the properties of $\Lambda^{\mathcal{DM}}$ and $\triangle^m_{D_s^t}$ and the relation of these operations with the minimal freeness operations. We also have that $extend^m(D_s^t, M_s^{*t}) =^{\mathcal{M}} extend^m(D_c^t, M_c^{*t}) \Lambda^{\mathcal{M}} \alpha^{\mathcal{M}}(C)$.

8.4.3 Procedure-entry

Procedure-entry computes the abstract constraint at the beginning of a clause used to resolve a procedure call, given the abstract call constraint of the call.

Definition 8.4.6 (Procedure-entry)
Let $((D_c^o, M_c^{*o}), (D_c^n, M_c^{*n}))$ be the abstract call constraint of a procedure call $p(Y_1, \ldots, Y_k)$ and let $p(Z_1, \ldots, Z_k)$ be the head of a clause[4] used to resolve the call. Then, the abstract constraint at the beginning of the clause is $AC_{in} = AC_{entry}\rho$ with $AC_{entry} = ((D_{in}^o, M_{in}^{*o}), (\emptyset, \emptyset))$ and $(D_{in}^o, M_{in}^{*o}) = \exists^{\mathcal{DM}}_{\{Y_1,\ldots,Y_k\}}(D_c^t, M_c^{*t})$ and $\rho = \{Y_1 \leftarrow Z_1, \ldots, Y_k \leftarrow Z_k\}$ (renaming).

It follows immediately from this definition that $(D_{in}^t, M_{in}^{*t}) = (\exists^{\mathcal{DM}}_{\{Y_1,\ldots,Y_k\}}(D_c^t, M_c^{*t}))\rho$.

8.4.4 Procedure-exit

Procedure-exit computes the abstract success constraint of a procedure call, given its abstract call constraint and the abstract constraints at the end of each clause used to resolve the call.

Definition 8.4.7 (Procedure-exit)
Let $((D_c^o, M_c^{*o}), (D_c^n, M_c^{*n}))$ be the abstract call constraint of a call $p(Y_1, \ldots, Y_k)$. Let the head of each clause Cl_i defining p/k be of the form $p(Z_1, \ldots, Z_k)$ $(1 \leq i \leq m)$.
Let $((D_i^o, M_i^{*o}), (D_i^n, M_i^{*n}))$ be the abstract constraint at the end of clause Cl_i and let $(D_{exit}^n, M_{exit}^{*n}) = lub^{\mathcal{DM}}(\{\ldots, (\exists^{\mathcal{DM}}_{\{Z_1,\ldots,Z_k\}}(D_i^n, M_i^{*n}))\rho^{-1}, \ldots\})$ with $\rho^{-1} = \{Z_1 \leftarrow Y_1, \ldots, Z_k \leftarrow Y_k\}$ (renaming). The abstract success constraint $((D_s^o, M_s^{*o}), (D_s^n, M_s^{*n}))$ of the call is defined as :

1. *$D_s^n = D_s^t \setminus D_c^o$ where D_s^t is obtained by performing procedure-exit on $Con^{\mathcal{D}}$ [43, 41] (D_s^t is the set of all variables that are definite after interpretation of the procedure call);*
2. *$D_s^o = D_c^o$;*
3. *$M_s^{*n} = M_c^{*n} \ \triangle^m_{D_s^t} M_{exit}^{*n}$;*
4. *$M_s^{*o} = reduce^m(D_s^t, M_c^{*o}) \cup (reduce^m(D_s^t, M_c^{*o}) \oplus^{\mathcal{M}} reduce^m(D_s^t, M_{exit}^{*n}))$*
$= reduce^m(D_s^t, M_c^{*o}) \cup (M_c^{*o} \oplus^m_{D_s^t} M_{exit}^{*n})$.

[3] One could first compute $\alpha^{\mathcal{DM}}(C) = (D', M'^*)$ and use M'^* instead of $\alpha^{\mathcal{M}}(C)$. However, reduction with D_s^t is still needed afterwards. So it is computationally more efficient to use $\alpha^{\mathcal{M}}(C)$ and then to perform reduction with D_s^t at once.

[4] The clause is assumed to be renamed apart from the variables in the abstract call constraint.

As in the case of the (minimal) freeness and $\mathcal{DF}$ abstractions, part of the information is recomputed at procedure-exit.

It can be shown that $(D_s^t, M_s^{*t}) =^{\mathcal{DM}} (D_c^t, M_c^{*t}) \Lambda^{\mathcal{DM}} (D_{exit}^n, M_{exit}^{*n})$, based on the definitions of $\Lambda^{\mathcal{DM}}$ and Definition 8.4.7. Restricting attention to the freeness part of the abstractions, we have that $M_s^{*t} =^{\mathcal{M}} M_c^{*t} \ \triangle_{D_s^t}^m \ M_{exit}^{*n}$. So the safety of procedure-exit is based on the properties of $\Lambda^{\mathcal{DM}}$ and $\triangle_{D_s^t}^m$ and the relation of these operations with the minimal freeness operations. Also, $extend^m(D_s^t, M_s^{*t}) =^{\mathcal{M}} extend^m(D_c^t, M_c^{*t}) \Lambda^{\mathcal{M}} extend^m(D_{exit}^n, M_{exit}^{*n})$.

8.4.5 Efficiency considerations

The efficiency considerations are similar to those of the $\mathcal{F}$, $\mathcal{M}$ and $\mathcal{DF}$ abstractions.

8.4.5.1 Time efficiency

To avoid the recomputation at procedure-exit, an alternative definition of a compound abstract constraint (and the associated abstract operations) can be formulated. The idea is the same as in the case of the freeness abstraction (Section 5.4.5.1) : in the alternative definition, the combination of information passed down at procedure-entry with local information gathered during the procedure execution is added to the n-component of a compound abstract constraint rather than to its o-component. This time-efficient approach may cause some loss of precision.

8.4.5.2 Space efficiency

Compound abstract constraints (both with the original and with the time-efficient approach) can be reduced by ensuring that the M^{*o} component does not contain sets that appear in M^{*n} or that are non-minimal with respect to M^{*t}. For example, the definition of abstract interpretation of a constraint then becomes :

Definition 8.4.8 (Abstract interpretation of a constraint (space-optimised)) *Let $C \in SCons$ and let $((D_c^o, M_c^{*o}), (D_c^n, M_c^{*n}))$ be the abstract call constraint of C. The abstract success constraint $((D_s^o, M_s^{*o}), (D_s^n, M_s^{*n}))$ of C is defined as :*

1. *$D_s^n = D_s^t \setminus D_c^o$ where D_s^t is obtained by abstract interpretation of C on $Con^{\mathcal{D}}$ [43, 41] (D_s^t is the set of* all *variables that are definite after abstract interpretation of C);*
2. *$D_s^o = D_c^o$.*
3. *$M_s^{*n} = M_c^{*n} \ \triangle_{D_s^t}^m \alpha^{\mathcal{M}}(C)$;*
4. *$M_s^{*o} = M_s^{*t} \setminus M_s^{*n}$ with $M_s^{*t} = M_c^{*t} \Lambda^{\mathcal{DM}} \alpha^{\mathcal{M}}(C)$.*

The definition of the other abstract operations is adjusted in a similar way. The order relation must be redefined as follows : $((D_1^o, M_1^{*o}), (D_1^n, M_1^{*n})) \leq^{\mathcal{DF}} ((D_2^o, M_2^{*o}), (D_2^n, M_2^{*n}))$ iff $D_1^o \supseteq D_2^o, D_1^n \supseteq D_2^n$, $extend^m(D_1^o \setminus D_2^o, M_1^{*o}) \subseteq close(M_2^{*t})$ *and* $extend^m(D_1^n \setminus D_2^n, M_1^{*n}) \subseteq close(M_2^{*n})$.

It can be shown that no information is lost by performing the optimisation (cf. Section 6.4.5.2).

The reduction optimises space. Also the effect on time efficiency is expected to be beneficial. The analysis time may decrease due to the computation with smaller abstract constraints.

Moreover, in the space-optimised definitions only $\Lambda^{\mathcal{M}}$ is used and no longer $\oplus^{\mathcal{M}}$; the latter is computationally more expensive (cf. Definitions 6.3.2 and 6.3.3). Also, computing with M^{*l} instead of M^{*o} may imply computing with a smaller abstract constraint since more minimisation is possible.

8.5 Examples

We illustrate the $\mathcal{DM}$ analysis on the same program examples as used in the previous chapters.

First of all, consider the following simple program involving only numerical constraints. We assume that the program is called with all variables being free (i.e. starting with the empty constraint store).

?- $(AC_0)\ Z = 1,\ (AC_1)\ p(X, Y, Z, T)\ (AC_2)$.

$p(X, Y, Z, T) \leftarrow (AC_3)\ X = Y + Z\ (AC_4)$.
$p(X, Y, Z, T) \leftarrow (AC_5)\ X = Y + T\ (AC_6)$.

The compound $\mathcal{DM}$ abstract constraints (not space-optimised) derived at each program point are the following :

$AC_0 = ((\emptyset, \emptyset),\ (\emptyset, \emptyset))$
$AC_1 = ((\emptyset, \emptyset),\ (\{Z\}, \emptyset)$
$AC_2 = ((\emptyset, \emptyset),\ (\{Z\}, \{\{X, Y\}, \{X, Y, T\}\})$

$AC_3 = ((\{Z\}, \emptyset),\ (\emptyset, \emptyset))$
$AC_4 = ((\{Z\}, \emptyset),\ (\emptyset, \{\{X, Y\}\}))$

$AC_5 = ((\{Z\}, \emptyset),\ (\emptyset, \emptyset))$
$AC_6 = ((\{Z\}, \emptyset),\ (\emptyset, \{\{X, Y, T\}\}))$

Space-optimisation does not yield further improvements for this example. All AC_i with $i \geq 1$ indicate that Z is definite. The $\mathcal{DM}$ abstract constraints (AC_2, AC_4, AC_6) are more compact than the abstract constraints obtained with the $\mathcal{F}$ or $\mathcal{M}$ analysis. For this particular example, no improvements are obtained with respect to the $\mathcal{DF}$ analysis.

The second example involves both numerical and unification constraints. Again, we assume that the program starts off with the empty constraint store.

?- $(AC_0)\ Z = 1,\ (AC_1)\ p(X, Y, Z, T)\ (AC_2)$.

$p(X, Y, Z, T) \leftarrow (AC_3)\ X = f(Z)\ (AC_4)\ , Z + T = 0\ (AC_5)$.
$p(X, Y, Z, T) \leftarrow (AC_6)\ X = g(Y)\ (AC_7)\ , Y - T = 0\ (AC_8)$.

The compound abstract constraints derived at each program point are :

$AC_0 = ((\emptyset, \emptyset),\ (\emptyset, \emptyset))$
$AC_1 = ((\emptyset, \emptyset),\ (\{Z\}, \emptyset))$
$AC_2 = ((\emptyset, \emptyset),\ (\{Z\}, \{\{T\}, \{X\}, \{X, Y\}, \{Y, T\}\}))$

$AC_3 = (\{\{\{Z\}\}, \emptyset\}, (\emptyset, \emptyset))$
$AC_4 = (\{\{\{Z\}\}, \emptyset\}, (\{X\}, \emptyset))$
$AC_5 = (\{\{\{Z\}\}, \emptyset\}, (\{X, T\}, \emptyset))$

$AC_6 = (\{\{\{Z\}\}, \emptyset\}, (\emptyset, \emptyset))$
$AC_7 = (\{\{\{Z\}\}, \emptyset\}, (\emptyset, \{\{X\}, \{X, Y\}\}))$
$AC_8 = (\{\{\{Z\}\}, \emptyset\}, (\emptyset, \{\{X\}, \{X, Y\}, \{Y, T\}, \{X, T\}\}))$

Space-optimisation does not yield any further improvements. Note that the $\mathcal{DM}$ abstract constraints are certainly much smaller than the $\mathcal{F}$ abstract constraints and are still more compact than (or, if not more compact, at least as compact as) the corresponding $\mathcal{DF}$ or $\mathcal{M}$ ones.

The mode information derivable from the AC_i is the same as for the $\mathcal{DF}$ abstraction (Section 7.5). No information is lost in the minimisation process.

The last example is the well-known sumlist program, which involves mixed constraints. It defines the relation between a list L and the sum S of the list elements. We assume that the program is called with the first argument being definite and the second argument free.

?- (AC_0) *sumlist*$(List, Sum)$ (AC_1).

sumlist(L, S) ←
    (AC_2) $L = [\,]$,
    (AC_3) $S = 0$ (AC_4).
sumlist(L, S) ←
    (AC_5) $L = [H \mid T]$,
    (AC_6) $S = H + S1$,,
    (AC_7) *sumlist*$(T, S1)$ (AC_8).

The $\mathcal{DM}$ abstract constraints obtained at each program point are shown below. For simplicity, we only present the $AC_i^t = (D_i^t, M_i^{*t})$ instead of showing the components AC_i^o and AC_i^n. Recall that the AC_i^t are sufficient to derive mode and dependency information.

$AC_0^t = (\{List\}, \emptyset)$
$AC_1^t = (\{List, Sum\}, \emptyset)$

$AC_2^t = (\{L\}, \emptyset)$
$AC_3^t = (\{L\}, \emptyset)$
$AC_4^t = (\{L, S\}, \emptyset)$

$AC_5^t = (\{L\}, \emptyset)$
$AC_6^t = (\{L, H, T\}, \emptyset)$
$AC_7^t = (\{L, H, T\}, \{\{S, S1\}\})$
$AC_8^t = (\{L, H, T, S, S1\}, \emptyset)$

Here, the results are the same as for the $\mathcal{DF}$ abstraction. The abstract constraints are much more compact than the ones obtained with the $\mathcal{F}$ analysis. Compared with the $\mathcal{M}$ analysis, mostly definiteness information is inferred now such that the freeness part is reduced to the empty set (except for AC_7^t).

More examples can be found in Appendix A.

Chapter 9

Extensions

In this chapter we discuss some extensions to the analyses presented so far. These extensions aim at enhancing the analyses in different respects.

A first extension concerns the treatment of non-normalised programs. Allowing non-normalised programs affects the practicality of the analysis (it no longer requires a translation of the user program to normal form and back), but also the precision and efficiency. However, it does not enhance the power of the analysis in the sense that it still captures the same program properties.

Although it is difficult to make a clear classification, a second group of extensions can be viewed as altering the analysis by capturing additional properties (possibly even implying a change of the form of the abstractions). A first extension of this kind concerns the treatment of disequations and inequalities. So far, only a rough abstraction of these constraints was performed. We discuss several ways to improve precision, thereby possibly changing the kind of information extracted from the constraints. Secondly, we describe how non-linear constraints (or passive constraints in general) can be handled. Again, several alternatives are possible of which some affect the form of the abstraction. A third extension consists of capturing definite/possible failure. This extension is particularly important with regard to the constraint reordering optimisation. Finally, the mode *free* can be refined by adding a limited form of type information to distinguish between free variables that are untyped and free variables that are numerical.

The extensions above do not change the language considered (being CLP(H,N)). A last part of this chapter describes how to extend the analyses to deal with other constraint domains besides H and N.

9.1 Analysis of non-normalised programs

In non-normalised programs, the arguments of literals and clause heads are no longer distinct variables but can be any terms. Moreover, it is no longer required that a composite primitive constraint involving both Herbrand and numerical functors is normalised, for example $X = f(Y + 1, T)$ no longer has to be translated into $X = f(V, T), V - Y = 1$. So, to extend the analysis to deal with non-normalised programs, the abstraction of a constraint and the abstract procedure-entry and procedure-exit operations (and their safety conditions) have to be redefined.

First, we present the adapted framework description, as given by García de la Banda and Hermenegildo in [43, 41]. Afterwards, as an example, we redefine the necessary operations for the freeness abstraction. Recall that additional complexity arises compared with the general description since *compound* abstract constraints are used. Similar changes apply in case of the improved freeness abstractions ($\mathcal{M}$, $\mathcal{DF}$ and $\mathcal{DM}$). Finally, the impact of normalisation or non-normalisation is discussed.

9.1.1 Adaptation of the framework description

9.1.1.1 Abstraction of a constraint

Let $C \in Cons$ be a constraint containing one or more composite primitive constraints. The abstraction of C during program analysis is computed by *at that point* transforming C to C' in which each composite constraint is replaced by its normal form (i.e. a conjunction of primitive unification and numerical constraints). For example, $C \equiv X = f(Y+1, T) \wedge Y+Z = 0$ is transformed into $C' \equiv X = f(V, T) \wedge V-Y = 1 \wedge Y+Z = 0$. Then $AC' = \alpha(C')$ is computed using the abstraction function of the normalised analysis. Afterwards, AC' is projected onto the variables of C : $\alpha(C) = \overline{\exists}_{vars(C)} AC'$.
The difference with the normalised analysis is that the normalisation is now done *during* the analysis and that the introduction of normalisation variables (such as the variable V in the example above) is only *local*.

9.1.1.2 Procedure-entry$(A, AC_c) = \{AC^1_{in}, \ldots, AC^m_{in}\}$

The procedure-entry operation takes as input an atom A and its abstract call constraint AC_c. For each clause[1] $Cl^j : H^j \leftarrow B^j$ defining A, it computes the abstract constraint AC^j_{in} to be attached to the first program point in Cl^j. The computation consists of two steps. The first step, i.e. computing AC_{entry}, is the same as for the normalised analysis: $AC_{entry} = \overline{\exists}_{vars(A)} AC_c$. Secondly, AC^j_{in} can be computed from AC_{entry} as follows : $AC^j_{in} = \overline{\exists}_{vars(Cl^j)}(AC_{entry} \wedge \alpha(A = H^j))$. The expression $A = H^j$ is an abbreviation for the conjunction of equations between the corresponding arguments of A and H^j. In some cases (cf. the freeness analysis in Section 9.1.2), the computation of AC^j_{in} may be somewhat more complex than applying abstract conjunction as such, in order to get more precise results (in the freeness analysis, abstract conjunction is nevertheless applicable on components of the abstract constraints).
SAFETY CONDITION :

1. For each $CS_c \in Con^c$ such that $CS_c \leq^c \gamma(AC_c)$ must hold that $\overline{\exists}_{vars(A)} CS_c \leq^c \gamma(AC_{entry})$.
2. For each $CS_c \in Con^c$ such that $CS_c \leq^c \gamma(AC_c)$ (so $\overline{\exists}_{vars(A)} CS_c \leq^c \gamma(AC_{entry})$) must hold that $\overline{\exists}_{vars(Cl^j)}(CS_c \wedge (A = H^j)) \leq^c \gamma(AC^j_{in})$.

Note that the safety of procedure-entry follows from the safety of abstract conjunction and abstract projection.

[1] Cl^j is assumed to be renamed apart from the variables in A and AC_c.

9.1.1.3 Procedure-exit$(A, AC_c, \{H^1, \ldots, H^m\}, \{AC^1_{out}, \ldots, AC^m_{out}\}) = AC_s$

Procedure-exit is applied when the final abstract constraint AC^j_{out} has been derived for each clause $Cl^j : H^j \leftarrow B^j$ defining A. It takes as input an atom A, its abstract call constraint AC_c, the heads H^j of the clauses defining A and the set of abstract constraints $\{AC^1_{out}, \ldots, AC^m_{out}\}$. It computes the abstract success constraint AC_s of A. In a first step, it computes $AC_{exit} = upp(\{AC^1_{exit}, \ldots, AC^m_{exit}\})$ where $AC^j_{exit} = \overline{\exists}_{vars(A)}(AC^j_{out} \wedge \alpha(A = H^j))$. The second step is the so called *extension* step which combines AC_{exit} and AC_c to obtain AC_s : $AC_s = AC_c \wedge AC_{exit}$; in some cases, a more complex variant of this is required in order to obtain sufficient precision (cf. freeness analysis in Section 9.1.2).

SAFETY CONDITION :

1. For each $CS^j_{out} \in Con^c$ such that $CS^j_{out} \leq^c \gamma(AC^j_{out})$ must hold that $CS_{exit} \leq^c \gamma(AC_{exit})$ with $CS_{exit} = \overline{\exists}_{vars(A)}(CS^j_{out} \wedge (A = H^j))$ and $1 \leq j \leq m$.
2. For each pair $CS_c, CS_{exit} \in Con^c$, such that $CS_c \leq^c \gamma(AC_c)$ and $CS_{exit} \leq^c \gamma(AC_{exit})$ and there exists a $CS_{local} \in Con^c$ over $vars(A)$ with[2] $CS_{exit} = (\overline{\exists}_{vars(A)} CS_c) \wedge CS_{local}$, must hold that $CS_c \wedge CS_{exit} \leq^c \gamma(AC_s)$.

Note that the safety of procedure-exit follows from the safety of abstract conjunction and abstract projection and the properties of *upp*.

9.1.2 Example : non-normalised freeness analysis

As an example, we define procedure-entry and procedure-exit for the non-normalised freeness analysis. Adjusting the abstraction of a constraint is exactly the same as in the general case mentioned above.

Definition 9.1.1 (Procedure-entry)
Let (AC^o_c, AC^n_c) be the abstract call constraint of a procedure call $p(t_1, \ldots, t_k)$ and let $p(s_1, \ldots, s_k)$ be the head of a clause[3] used to resolve the call; $t_1, \ldots, t_k$ and $s_1, \ldots, s_k$ are general terms. Then, the abstract constraint at the beginning of the clause is defined as $AC_{in} = (\overline{\exists}^{\mathcal{F}}_{vars(s_1,\ldots,s_k)}(AC^o_{entry} \wedge^{\mathcal{F}} \alpha^{\mathcal{F}}(t_1 = s_1 \wedge \ldots \wedge t_k = s_k)), \emptyset)$ with $AC_{entry} = (AC^a_{entry}, AC^n_{entry}) = (\overline{\exists}^{\mathcal{F}}_{vars(t_1,\ldots,t_k)} AC^t_c, \emptyset)$.

Definition 9.1.2 (Procedure-exit)
Let (AC^o_c, AC^n_c) be the abstract call constraint of a call $p(t_1, \ldots, t_k)$. Let the head of each clause Cl_i defining p/k be of the form $p(s_1, \ldots, s_k)$ $(1 \leq i \leq m)$. Let (AC^o_i, AC^n_i) be the abstract constraint at the end of Cl_i.
Then $AC^n_{exit} = lub^{\mathcal{F}}(\{\ldots, \overline{\exists}^{\mathcal{F}}_{vars(t_1,\ldots,t_k)}(AC^n_i \wedge^{\mathcal{F}} \alpha^{\mathcal{F}}(t_1 = s_1 \wedge \ldots \wedge t_k = s_k)), \ldots\})$ and the abstract success constraint (AC^o_s, AC^n_s) of the call is defined as :

[2] CS_{local} is the set of all possible constraints – matched back to $vars(A)$ via $A = H^j$ and projected onto $vars(A)$ – that may be gathered during execution of Cl^j (each constraint is gathered along some OR-branch of the concrete AND-OR tree). Alternatively, the condition $CS_{exit} = (\overline{\exists}_{vars(A)} CS_c) \wedge CS_{local}$ can be expressed as follows (without using CS_{local}) : $CS_{exit} \Rightarrow \overline{\exists}_{vars(A)} CS_c$.

[3] The clause is assumed to be renamed apart from $vars(AC^t_c)$.

1. $AC_s^o = AC_c^o \cup (AC_c^o \oplus^{\mathcal{F}} AC_{exit}^n)$
2. $AC_s^n = AC_c^n \wedge^{\mathcal{F}} AC_{exit}^n$

In this definition, $\exists^{\mathcal{F}}_{vars(t_1,\ldots,t_k)}(AC_i^n \wedge^{\mathcal{F}} \alpha^{\mathcal{F}}(t_1 = s_1 \wedge \ldots \wedge t_k = s_k))$ could be replaced by $\exists^{\mathcal{F}}_{vars(t_1,\ldots,t_k)}(\,(\exists^{\mathcal{F}}_{vars(s_1,\ldots,s_k)} AC_i^n) \wedge^{\mathcal{F}} \alpha^{\mathcal{F}}(t_1 = s_1 \wedge \ldots \wedge t_k = s_k)\,)$. The latter uses one more abstract projection. However, a smaller abstract constraint is obtained before applying abstract conjunction. Since the efficiency of abstract conjunction strongly depends on the size of the abstract constraints, the overall computation (with the additional projection) may still be faster. This approach is taken in our implementation.

9.1.3 Impact of (non-)normalisation

Dealing with non-normalised programs allows to improve the precision of the derived information. At least, this is the case when abstracting unification or composite constraints (not when considering numerical constraints). The improvement of the precision is illustrated by the following example.

Example 9.1.1
Consider the following program fragment :

$\ldots, p(f(X,Y)), \ldots$
$p(f(A,B)) \leftarrow q(A,B).$

Assume that before calling $p/1$ the variable X is free and Y has mode any (for the minimal freeness abstraction this implies that the abstract call constraint[4] AC^t of p is $\{\{Y\}\}$). The non-normalised analysis yields that before the call to $q/2$, i.e. after the call-head matching $f(X,Y) = f(A,B)$, A is free and B has mode any (the abstract call constraint for q is $\{\{B\}\}$).
Now consider the normalised analysis, starting from the normalised program

$\ldots, V_1 = f(X,Y),\ p(V_1), \ldots$
$p(V_2) \leftarrow V_2 = f(A,B),\ q(A,B).$

Here, V_1 and V_2 are fresh variables. Before $V_1 = f(X,Y)$, V_1 and X are known to be free and Y has mode any. After analysing this unification (before the call to p) V_1 and Y have mode any and X is free. So the minimal abstract call constraint of p is $\{\{V_1\},\{Y\}\}$, or projected onto the variables of p we have $\{\{V_1\}\}$. Performing procedure-entry yields that V_2 has mode any and A and B are free before $V_2 = f(A,B)$ (the abstract constraint is $\{\{V_2\}\}$). After performing the unification, i.e. just before the call to q, we obtain that V_2, A and B all have mode any (the abstract call constraint for q is $\{\{V_2\},\{A\},\{B\}\}$). This is less precise than the call-pattern for q obtained with the non-normalised analysis.

In a non-normalised analysis more precise call patterns can be expressed, compared to a normalised analysis : one can express that an argument is a partially instantiated structure of a particular form, containing a number of free components.

[4] We do not consider the old and new components of the abstract constraint separately as it is irrelevant here.

Example 9.1.2
The directive

$$:\text{- } entry(p(f(X1,X2,X3)),\ [mode(X1,\mathrm{f}),\ mode(X2,\mathrm{d}),\ mode(X3,\mathrm{f})]).$$

indicates that $p/1$ is called with a term $f(X1,X2,X3)$ as argument, where $X1$ and $X3$ are free variables and $X2$ is a definite variable. It is not possible to express this call pattern in a normalised analysis, since the arguments of predicate calls (also the entry call) must be distinct variables. One could introduce an auxiliary entry predicate main/3 of the form

$$main(X1,X2,X3) : -V = f(X1,X2,X3), p(V).$$
$$\text{with}\quad :\text{-}entry(main(X1,X2,X3),\ [mode(X1,\mathrm{f}),\ mode(X2,\mathrm{d}),\ mode(X3,\mathrm{f})]).$$

However, this gives rise to the call pattern $p(\mathrm{a})$, so the information on the structure of the argument and the mode of its components is lost.

Secondly, we consider the impact of normalisation on the efficiency of the analysis. Here, it is not clear whether the normalised or non-normalised analysis is more efficient. Experimental results are given in Chapter 11. However, we can already make some observations. First of all, the non-normalised programs involve less variables compared to the normalised ones. Even if auxiliary variables are introduced, which is the case when abstracting composite constraints, the addition of these variables is only *local* and has no effect on the further computation. The size of the abstract constraints depends on the number of variables in a clause or query, so abstract constraints will be smaller in case of the non-normalised analysis. Recall that this size strongly influences the efficiency of the abstract operations (especially abstract conjunction).
A second point in favour of the non-normalised analysis is that the extra time needed for normalisation is avoided (except for the local normalisation of composite constraints), although this constitutes only a small portion of the overall analysis time[5].
Thirdly, consider the treatment of recursive calls. In the non-normalised analysis, a recursive call $p(t_1,\ldots,t_k)$ with abstract entry constraint AC_{entry} is compared with an ancestor call $p(s_1,\ldots,s_k)$ with entry constraint AC'_{entry}. An additional condition to be checked besides the relation between AC_{entry} and AC'_{entry} (cf. case 3 in Algorithm 3.2.1) is whether $p(t_1,\ldots,t_k)$ and $p(s_1,\ldots,s_k)$ are renamings of each other. Only then, the recursive call may use the results of the ancestor call. Otherwise, it is considered as a new call and is analysed as usual (case 2 in Algorithm 3.2.1)[6]. Note that termination is still ensured since the number of syntactically different calls appearing in a program is limited. In the normalised analysis, two predicate calls are always a renaming of each other as their arguments are just distinct variables. This implies that the number of calls analysed during the non-normalised analysis (in other words, the size of the AND-OR graph) is usually larger than in the normalised case. In this respect, the non-normalised analysis may be less efficient (both with respect to time and space) than the normalised one.
Finally, one could note that non-normalised programs contain fewer program points compared to the normalised ones. However, the abstract conjunction operations that are

[5]In our implementation of the analysers for normalised programs, we assume that programs have been normalised in a pre-processing phase; so the normalisation time is not included into the analysis time.

[6]This condition could be relaxed by setting a weaker "similarity" criterion of predicate calls than syntactic identity (up to renaming). This is outside the scope of this text.

performed during abstract interpretation of the explicit constraints in the normalised case are performed during procedure-entry and procedure-exit in the non-normalised case.

A third aspect besides precision and efficiency of the analysis is its practicability and generality. In this respect the non-normalised analysis is better than the normalised one. First of all, it requires no preprocessing step to translate the user program to normal form and back (since the user wants to get the annotated version of his *original* program as output). Secondly, the non-normalised analysis system is more general since it can be applied to normalised programs also. However, this may incur some time overhead compared to the normalised analysis. The reason is that procedure-entry and procedure-exit in the former analysis involve abstract conjunction and abstract projection, which are more costly than the renaming operation in the normalised analysis.

A last issue is the impact of normalisation on program annotation. One of the considered program optimisations is the shifting of numerical constraints over clause literals towards the end of the clause. In this respect, annotation is easier if the constraint appears explicitly within the clause body rather than hidden in a head argument. Therefore, it is advantageous to set up numerical constraints explicitly, especially since no precision is lost because of this normalisation.

9.2 Analysis of linear disequations and inequalities

In the analyses presented so far, a single linear disequation or inequality is precisely abstracted as the set of its variables. However, when abstracting a *conjunction* of primitive linear equations, disequations and inequalities (called a *generalised linear constraint* [69] in the sequel), precision is lost since all constraints are treated as equations, i.e. as symmetrical and transitive relations. For example, for $C \equiv X > Z \wedge Y > Z$ we have that $\alpha^{\mathcal{F}}(C) = close(\{\{X,Z\},\{Y,Z\},\{X,Y\}\})$ although there is no concrete dependency between X and Y.

It is straightforward to adjust the definition of the abstraction function (Definition 5.2.10) towards generalised linear constraints, using their solved form as defined in [69].

Definition 9.2.1 (Abstraction of generalised linear constraints)
Let C be a generalised linear constraint. Let

$$W = \left\{ \{X_1,\ldots,X_n\} \;\middle|\; \begin{array}{l}(a_1X_1+\cdots+a_nX_n \diamond b) \in sform(C) \\ \text{for any } sform(C),\ \diamond \in \{=,\neq,>,\geq\}\end{array} \right\}.$$

Then, $\alpha^{\mathcal{F}}(C) = close(W)$ where $close(W)$ denotes the closure under union of W.

For the above example, the new definition yields $\alpha^{\mathcal{F}}(C) = close(\{\{X,Z\},\{Y,Z\}\})$, which is precise. However, this still doesn't solve the loss of precision at abstract conjunction. When joining two abstract constraints, there is not enough information in the abstractions to obtain a precise result. For example, consider $C_1 \equiv X > Z$ and $C_2 \equiv Y > Z$; $\alpha^{\mathcal{F}}(C_1) = \{\{X,Z\}\}$ and $\alpha^{\mathcal{F}}(C_2) = \{\{Y,Z\}\}$. Then $\alpha^{\mathcal{F}}(C_1) \wedge^{\mathcal{F}} \alpha^{\mathcal{F}}(C_2) = close(\{\{X,Z\},\{Y,Z\},\{X,Y\}\})$ although $C_1 \wedge C_2$ does not entail a dependency between X and Y.

The only way to increase precision is to change the form of the abstraction. It should at least keep track of the type of constraint ($=, \neq, >$ or $\geq$) and the sign of the coefficients. An abstract domain of this kind, called *LSign*, is presented by Marriott and Stuckey [86, 87]. The domain as described in [87] does only deal with equations and inequalities, not disequations. In the sequel, we mention how it can be extended towards generalised linear constraints. The abstraction of a generalised linear constraint is obtained by replacing the coefficients by their signs (a strictly positive coefficient is replaced by $\oplus$, a strictly negative one by $\ominus$ and zero is kept as such; a special symbol $\top$ represents a coefficient for which there is no precise information and hence may be positive, negative or zero). For example, $C \equiv X - Y > 0 \wedge Y - Z > 5$ is abstracted as $\alpha(C) = \oplus X + \ominus Y > 0 \wedge \oplus Y + \ominus Z > \oplus$. To improve precision when dealing with primitive equations, the abstraction of equations is extended with multiplicity information ($=$ represents exactly one concrete equation, $=^+$ represents one or more concrete equations, $=^?$ represents zero or one concrete equation and $=^*$ represents any number of concrete equations; for more details we refer to [87]). Abstract conjunction consists of taking the union of two abstractions, with an additional operation to deal with multiplicity information. Abstract projection in [87] is basically an abstraction of the Gauss-Jourdan algorithm (for certain types of equation) or of the Fourier-Motzkin algorithm (for general equations and inequalities) to perform variable elimination. To deal with disequations one could use an abstraction of the algorithms for variable elimination in generalised linear constraints presented in [52]. The first approach described there can be integrated into the Fourier algorithm for variable elimination in inequalities. It is not yet clear what is the effect on the complexity of the analysis. A less obvious operation is the upper bound. If at each point in the analysis all possible (entailed) concrete constraints should be captured (as is the case for freeness analysis), the upper bound operation should be defined as a set of the abstractions of the different clauses. Although in theory this does not threaten termination (since the number of constraints and the number of variables in a program is finite), this may give rise to large abstractions and therefore a suitable widening operation should be devised. In [87] this is not clearly elaborated. Another issue concerns the interaction between the numerical constraint part and the unification part. The elegant merging of information from the two parts that was used in our analyses presented so far can no longer be applied. Explicit interaction rules must be defined, increasing the complexity of the analysis. Moreover, some variable dependencies are established by a combination of numerical and unification constraints and do not belong to either the numerical or the unification part. For example, $X = f(A, B) \wedge A + B = T$ establishes the dependency $\{X, T\}$. In our view, it seems that dependencies of this kind cannot be modelled within the *LSign* analysis, unless an extra spanning component is added to the abstract constraint description. In summary, the LSign abstraction is strictly more accurate than our analyses presented so far, especially in the case of inequalities; however, the increase of precision most likely comes at a price of increased complexity and lower efficiency. Unfortunately, [87] does not report on an implementation of the LSign abstraction such that comparison with our analysers is not possible.

Although the information in the LSign analysis is already quite accurate, precision can still be improved by taking into account the *values* of the coefficients as long as possible – in other words, by keeping track of the concrete numerical constraints as long as possible. Of course, this will further increase the complexity of the analysis. The most difficult operation is again the upper bound, where a suitable widening (or abstraction) should be

applied.
To our knowledge, there exist only a few analyses that keep track of concrete constraints to some extent. Janssens, Bruynooghe and Englebert [60] present several abstractions for CLP(H,N), with increasing complexity and precision. The kernel abstraction is based on intervals and narrowing rules [73] having only a local impact. It can be extended (1) by applying the narrowing rules more globally and (2) by adding ordinal relationships (i.e. certain types of inequalities). The extended abstractions keep track of concrete numerical constraints. However, they compute an underestimation (i.e. a weakening) of the concrete set of constraints at each point. Hence it does not allow to infer all *possible* concrete dependencies needed in case of the freeness analysis. Another abstraction of numerical constraints that is briefly described in [86] and was first presented by Cousot and Halbwachs [28] is the convex hull abstraction. Again, this abstraction is not suitable for freeness analysis as it derives an underestimation of the concrete constraint store. Bagnara, Giacobazzi and Levi [3, 2] abstract numerical constraints by means of labelled digraphs. Nodes are labelled with a numerical expression, occurring in a numerical constraint of the program, or with an interval (if the analysis can no longer maintain a precise numerical expression); arcs are labelled with constraint symbols $(=, >, \ldots)$. Manipulating the digraphs is done using constraint inference techniques. It is not clear how the analysis deals with merging results of different clauses.

Finally, we describe another issue that has an impact on the abstraction of programs containing generalised linear constraints. It concerns using information about (future) redundancy of the constraints. A constraint is called redundant if it is entailed by each possible constraint store C that can be obtained just before that constraint. Then the constraint does not have to be considered by the constraint solver (in particular, it does not have to be added to the store). A constraint is future redundant (according to the restrictive definition of [63], not the generalised one of [86]) if it is entailed by the conjunction of the store C with the constraints to be added in the next cycle of the constraint engine. In this case, it is sufficient to check whether the constraint is consistent with the store; it does not have to be added to the store – neither during concrete nor during abstract computation. So the abstract success constraint AC_s of a (future) redundant constraint C is simply identical to its abstract call constraint AC_c. Avoiding the addition of (future) redundant constraints to the abstract store improves the efficiency of the analysis. Moreover, the results obtained are more precise, yielding more opportunities for program optimisation. Of course, the abstraction is only safe if also during concrete execution future-redundant constraints are not added to the store.

Example 9.2.1
Consider the simple sum/2 program that computes the sum S of the first N natural numbers.

$$
\begin{array}{l}
sum(N,S) \leftarrow N = 0, S = 0. \\
sum(N,S) \leftarrow \\
\qquad (AC_1)\ N \geq 1,\ (AC_2)\ S \geq N, \\
\qquad (AC_3)\ N = N1 + 1,\ (AC_4)\ S = S1 + N, \\
\qquad (AC_5)\ sum(N1, S1).
\end{array}
$$

The constraints $N \geq 1$ and $S \geq N$ are future redundant. Not adding these constraints to the store gives rise to the following (minimal) abstract constraints (assuming that the program is called with its first argument being possibly non-free and its second argument being free):

$AC_1 = \{\{N\}\}$;
$AC_2 = AC_1$; *the abstraction* $\{N\}$ *of* $N \geq 1$ *is not added again !*
$AC_3 = AC_2$; *the abstraction* $\{S, N\}$ *of* $S \geq N$ *is not added !*
$AC_4 = \{\{N\}, \{N1\}\}$; $AC_5 = \{\{N\}, \{N1\}, \{S, S1\}\}$

Whereas without the information on (future) redundancy we obtain less precise information:

$AC_1 = \{\{N\}\}$;
$AC_2 = AC_1$; *the abstraction* $\{N\}$ *of* $N \geq 1$ *is added a second time*
$AC_3 = \{\{N\}, \{S\}\}$; *the abstraction* $\{S, N\}$ *of* $S \geq N$ *is added*
$AC_4 = \{\{N\}, \{S\}, \{N1\}\}$; $AC_5 = \{\{N\}, \{S\}, \{N1\}, \{S1\}\}$

Information on (future) redundancy can be obtained via a separate program analysis. Several approaches have been described [63, 3, 86, 60]. Some of them [63, 3] use only a local analysis whereas the others [86, 60] are based on a global analysis and are therefore more powerful. Our analysers could easily be extended to accept input programs annotated with the (future) redundancy information.

9.3 Analysis of passive constraints

Passive constraints are constraints that are delayed during program execution until certain wake-up conditions, involving (a subset of) the variables in the constraints, are satisfied. In the CLP(H,N) languages that we consider (PrologIII, CLP($\mathcal{R}$)), non-linear constraints are passive constraints : they are delayed until they become linear. On the level of concrete semantics, integrating passive constraints is done by representing a state in the operational semantics as a tuple $\langle \mathcal{G}, C, S \rangle$ [55], where $\mathcal{G}$ is the current goal and C and S represent the current constraint store with C being the collection of active constraints and S being the collection of passive constraints. The standard conjunction operation is performed on S instead of on C; after each conjunction operation, an $\mathit{infer}(C, S) = (C', S')$ step is performed to join passive constraints that are activated to the active part C yielding C' and reducing S to S'.

When considering how to deal with passive constraints during abstract computation, at least three issues arise.

A first and fundamental question is whether the delay and wake-up of passive constraints are modelled at the abstract level or not. This involves trading off precision versus complexity of the resulting analysis. Modelling the delay and wake-up implies that abstract constraints are split up in two parts, an active part representing the active constraints and a passive part representing the passive constraints. An alternative method is to immediately abstract all the information in passive constraints without distinguishing this information from that provided by the active constraints (as though the constraints were active constraints) . This abstraction has to be safe with respect to all possible (future) activations of the passive constraint (explained further below). The latter approach is

simpler than the first one, since the abstract operations are less complex (there is only one component in an abstract constraint and there is no need for an abstract *infer* operation). We adopted this simple approach in our implemented analysers.

Secondly, when choosing to model the delay and wake-up of passive constraints, the issue is how to deal with that wake-up. Either one deals with all *possible* wake-ups of passive constraints at the abstract level, or one abstracts only *definite* wake-ups. An alternative formulation is : either one detects the earliest possible wake-up point of a passive constraint, or one detects some point at which the constraint is definitely woken. In the first case, the activation of a passive constraint at the abstract level may precede the activation at the concrete level and must not occur later. In the second case, the activation at the abstract level may follow the activation at the concrete level (certainly occur not sooner) or may even never occur. Which approach is safe depends on the type of analysis, i.e. on the kind of abstract properties that should be captured or –stated in another way – whether an upper or a lower approximation of the concrete constraint store should be computed. If a lower approximation must be computed, as is the case for the definiteness analysis, only *definite* activations of passive constraints need to be considered (this approach was taken by Hanus in [48]). Whereas if an upper approximation is needed, as for the freeness analysis, the *earliest possible* activation point of passive constraints needs to be considered. A related issue is termination. If the analysis has to detect the earliest possible wake-up point of a passive constraint, the abstract projection function has to leave enough information to ensure the correct wake-up behaviour. A straightforward way to guarantee this is to project only the abstract active constraints and to keep the passive part. In that case, an abstract constraint is no longer restricted to a finite number of variables (the variables of the clause or query) as it is in the original abstract interpretation framework. As a consequence, termination is not guaranteed and some new kind of widening should be introduced. This is related to the work described in [79]; this paper gives a simple denotational semantics and a generic global dataflow analysis algorithm which is based on this semantics. It deals with languages in which computation generally proceeds left-to-right but in which some calls (both atoms and constraints) are dynamically delayed until their arguments are sufficiently instantiated, a very similar case to that of the passive constraints.

Finally, we discuss what abstract information has to be extracted from a passive constraint, whether delay and wake-up are modelled or not. Conceptually, when a passive constraint is activated, it turns into an active one and is joined (via the *infer* function) to the active part[7]. At the abstract level however, activation of a passive constraint does not lead to a single concrete constraint but rather to a set of possible active constraints due to the fact that precise information on values of variables may not be available. Consider for example the passive constraint $X * Y = Z$ during freeness analysis; if X becomes possibly non-free, the constraint is activated, but two cases have to be distinguished : either $X = 0$ in which case the active constraint is $0 * Y = Z$, so Z is possibly non-free and Y is still free; or $X = n$ with n a non-zero number in which case the active constraint is $n * Y = Z$, so neither Y nor Z becomes non-free. If the abstraction provides no information on the value of X, as is the case in our freeness abstractions, both cases have to be considered. The abstraction of a passive constraint (without taking into account the environment information) should

[7]When delay and wake-up are not modelled, a passive constraint is considered as active at once.

therefore be such that conjunction of this abstract information with the active information (during the *infer* function) covers all possible actual situations that may occur after this conjunction. For example, for $X * Y = Z$ during freeness analysis a safe abstraction states that there is a possible dependency between X and Z and between Y and Z w.r.t. non-freeness propagation; during definiteness analysis a safe abstraction states that Z depends on both X and Y w.r.t. definiteness propagation (note that nothing more can be said about X or Y, since X is not necessarily definite if both Y and Z are definite as Y and Z may both be 0). Having more precise information on the possible values of variables, which could e.g. be provided by the analysis proposed in [60], could reduce the situations to be covered and thus improve the abstraction of passive constraints. If delay and wake-up are not modelled, the abstract information is joined immediately with the current (abstract) constraint store. Otherwise it is added to the passive abstract constraint part, with some extra information on the variables that determine the wake-up of the constraint. For example in case of $X * Y = Z$, the definiteness dependency of Z on both X and Y becomes active as soon as at least X or Y has become definite; during freeness analysis, the dependencies $\{X, Z\}$ and $\{Y, Z\}$ must be activated as soon as X or Y has become possibly non-free.

Let us now focus on the freeness abstraction. Several abstractions of non-linear constraints are possible. A safe but very rough abstraction of a primitive non-linear constraint c is $\alpha^{\mathcal{F}}(c) = \wp_{\emptyset}(vars(c))$ (hence considering each variable in $vars(c)$ as possibly non-free). A more precise approach is based on dividing $vars(c)$ into disjoint subsets and then constructing $\alpha^{\mathcal{F}}(c)$ from these subsets. We first present an algorithm to compute the disjoint subsets.

Algorithm 9.3.1 (varsets(c) = $\{S'_1, \ldots, S'_k\}$)
Let c be a primitive non-linear constraint of the form $t_1 + \ldots + t_n \Diamond b$ where b is a number, $\Diamond \in \{=, \neq, >, \geq\}$ and t_i $(1 \leq i \leq n)$ is a numerical term; at least one t_i is a non-linear term. Let $S_i = vars(t_i)$. The idea is to join the sets S_i that share at least one variable until one obtains sets $S'_1, \ldots, S'_k$ that are pairwise disjoint $(1 \leq k \leq n)$:

$SS := \{S_1, \ldots, S_n\}$; $varsets(c) := \emptyset$;
while $SS \neq \emptyset$ do
select a $S_i \in SS$; $SS := SS \setminus \{S_i\}$; $S' := S_i$;
for each $S'_j \in varsets(c)$ do
if $S'_j \cap S_i \neq \emptyset$
then begin $S' := S' \cup S'_j$; $varsets(c) := varsets(c) \setminus \{S'_j\}$ end;
$varsets(c) := varsets(c) \cup \{S'\}$

The idea behind joining together sets S_i (corresponding to subterms in the constraint) if they share a variable is that this variable can possibly be factored out of the associated subterms in the constraint. For example, $X * Y + X * Z - T = 5$ can be rewritten as $X * (Y + Z) - T = 5$; note that constraining X to 0 causes both the first and second subterm to disappear in the first form of the constraint, in the latter form this reduces to the disappearance of the first subterm $X * (Y + Z)$.

Definition 9.3.1 (Abstraction of a non-linear primitive constraint)
Let c be a non-linear primitive constraint and let $varsets(c) = \{S'_1, \ldots, S'_k\}$ be the set of

disjoint sets of variables computed using Algorithm 9.3.1. Then $\alpha^{\mathcal{F}}(c) = close(S'_1 \times \ldots \times S'_k)$ *where* $S'_1 \times \ldots \times S'_k = \{\{X_1, \ldots, X_k\} \mid X_1 \in S'_1, \ldots, X_k \in S'_k\}$ *(Cartesian product).*

Example 9.3.1
$\alpha^{\mathcal{F}}(X * Y = Z) = close(\{\{X, Z\}, \{Y, Z\}\})$*; constraining* X *or* Y *may cause* Z *to become non-free (*Z *becomes non-free if* $X = 0$ *or* $Y = 0$ *is added).*
$\alpha^{\mathcal{F}}(P1 = P * (1 + I/1200) - R) = close(\{\{P1, P, R\}, \{P1, I, R\}\})$.
$\alpha^{\mathcal{F}}(X - X * Y + T = 3) = close(\{\{X, T\}, \{Y, T\}\})$*; note that constraining* X *may cause* T *to become non-free (if* $X = 0$*) and also constraining* Y *may cause* T *to become non-free (if* $Y = 1$*). In terms of simplification of the original constraint, constraining* X *to* 0 *entails* $T = 3$*, since the first and second term disappear; similarly, constraining* Y *to* 1 *entails* $T = 3$ *(note that* $X - X * Y + T = 3$ *can also be written as* $X * (1 - Y) + T = 3$*).*

Although the above abstraction of non-linear constraints is already more precise, it can certainly be improved. One way to do this would be to design an asymmetric abstraction since the information to be captured is asymmetric (a fundamental point hereby is that the form of the abstraction is changed). For example considering $X * Y = Z$, X or Y can influence the non-freeness of Z but not vice versa; a more precise abstraction could state that X depends on both Y and Z, Y depends on both X and Z and Z depends on X or on Y. However, it is not straightforward to integrate such an abstraction with the abstraction of the other constraints. How the integration of the abstraction of Definition 9.3.1 can be done is described in the following algorithm.

Algorithm 9.3.2 (Abstraction of a constraint)
Let $C \in Cons$ *be a constraint possibly involving non-linear primitive constraints. Let* Lin *be the conjunction of the linear primitive constraints in* C *and* $NonLin$ *be the conjunction of the non-linear primitive constraints in* C*.*

1. *Compute* $sform(Lin)$ *(using the algorithms in Section 2.2).*
2. *For each primitive constraint in* $sform(Lin)$ *that is of the form* $X = n$ *with* n *a number: substitute* X *by* n *in* $NonLin$ *yielding* $NonLin'$*.*
3. *If* $NonLin' = NonLin$ *then goto step 4*
   else, for each primitive constraint c *in* $NonLin'$ *that has become linear due to step 2, remove* c *from* $NonLin'$ *and add it to* $sform(Lin)$*, resulting in a reduced* $NonLin''$ *and an extended* Lin''*; then continue with step 1 (replacing* Lin *with* Lin'' *and* $NonLin$ *with* $NonLin''$ *in the formulation of steps 1 and 2).*
4. *Given the final linear and non-linear part, compute the abstraction of the linear part (which is in solved form) using the abstraction function of Section 5.2.3.1; then, for each primitive constraint in the non-linear part, compute its abstraction using Definition 9.3.1 and join it with the abstraction computed so far using the abstract conjunction operation.*

9.4 Inferring definite and possible failure

For some applications, it may be necessary to capture at exactly which program points failure possibly/definitely occurs. For example, this information is indispensable for constraint and goal reordering and for parallelisation. The analyses presented in the previous

chapters capture only *definite* failure (also called definite unsatisfiability). They do not allow to infer whether failure *possibly* occurs at some point. Also, the analyses do not distinguish definite failure from unreachability, where unreachability means that a program point cannot be reached due to failure at a preceding point or due to an infinite loop of a preceding recursive call[8].

Below we develop an abstraction that captures failure information. In order to get more insight into this abstraction, we first give a precise description of the concrete collecting semantics, which incorporates the idea of local subcomputations as in the framework of Bruynooghe [6] (based on Local SLD instead of SLD [7]).

Consider a derivation starting from a collecting state $s_i = \langle l_i :: \ldots :: l_n, CS_i \rangle$ for a program P, where $l_i :: \ldots :: l_n$ is the current goal and CS_i is the set of possible current constraint stores ($::$ denotes concatenation, $CS_i \in \wp(Cons)$).

- If $CS_i = \emptyset$ or $CS_i = \{false\}$, then the successor state s_{i+1} is $s_{i+1} = \langle l_{i+1} :: \ldots :: l_n, \emptyset \rangle$ (no local subcomputation is started for l_i; also, failure is not propagated).

- Otherwise ($CS_i \neq \emptyset$ and $CS_i \neq \{false\}$),

  1. If $l_i \in Cons$, then $s_{i+1} = \langle l_{i+1} :: \ldots :: l_n, CS_{i+1} \rangle$ with $CS_{i+1} = (CS_i \setminus \{false\}) \wedge \{l_i\}$ (note : $CS_{i+1} = \{false\}$ if $sform(l_i) = false$ or $sform(l_i \wedge C) = false$ for each $C \in CS_i$, and $false \in CS_{i+1}$ if $sform(l_i \wedge C) = false$ for at least one $C \in CS_i$).

  2. If $l_i \in Atom$, then :
     if there exists no clause $h \leftarrow B$ in P such that l_i and h have the same predicate symbol p/k, then $s_{i+1} = \langle l_{i+1} :: \ldots :: l_n, \{false\} \rangle$; otherwise a local derivation is started :

     a. For each renaming $Cl^j : h^j \leftarrow b^j_1 :: \ldots :: b^j_m$ of a clause in P, such that l_i and h^j have the same predicate symbol p/k and $vars(Cl^j) \cap vars(s_i) = \emptyset$:
     let $CS_{entry} = \exists_{vars(l_i)}(CS_i \setminus \{false\})$ and $CS^j_{in} = \exists_{vars(Cl^j)}(CS_{entry} \wedge (l_i = h^j))$;
     then $s^j_{in} = \langle b^j_1 :: \ldots :: b^j_m, CS^j_{in} \rangle$ (s^j_{in} is the collecting state at the beginning of the clause Cl^j).

     b. The local derivation starting in s^j_{in} is developed.

     c. Finally, a final state $\langle \epsilon, CS^j_{out} \rangle$ (ϵ denotes the empty goal) is reached for each Cl^j. Let $CS_{exit} = \bigcup_j \exists_{vars(l_i)}(CS^j_{out} \wedge (l_i = h^j))$. Then $s_{i+1} = \langle l_{i+1} :: \ldots :: l_n, CS_{i+1} \rangle$ with $CS_{i+1} = (CS_i \setminus \{false\}) \wedge (CS_{exit} \setminus \{false\})$.

If CS in some state is $\{false\}$, it indicates *definite* failure caused by the last derivation step. If CS contains *false* but also at least one satisfiable constraint, then CS indicates *possible* failure. Finally, $CS = \emptyset$ denotes unreachability due to failure at a preceding point (not caused by the last derivation step) or to an infinite loop of a preceding recursive call. Definite/possible failure is not propagated from one program point to the following; *false* should only be part of a constraint set CS at some point if failure is caused by the *last* derivation step.

[8] In the literature, failure is often considered as a special case of unreachability. Here we choose to distinguish between the two.

The abstract domain for the original freeness abstraction[9] ($\mathcal{F}$) is extended with the special element *Afail*, corresponding to $\{false\}$ in the concrete domain. So, *Afail* denotes definite failure. The abstract constraint $\bot$ denotes unreachability (corresponding to $\emptyset$ in the concrete domain). Abstract constraints (different from $\bot$ and *Afail*) may now contain $\emptyset$, indicating possible failure[10].

Definition 9.4.1 (Abstract domain)
The abstract domain is $Con^{Fail} = \wp(\wp(Var)) \cup \{\bot, Afail\}$.

Definition 9.4.2 (Abstract computational order)
The order relation $\leq$ *is defined as follows :*

- $\bot \leq AC$ *for each* $AC \in Con^{Fail}$;
- $Afail \leq AC$ *for each* $AC \in Con^{Fail}$ *with* $\emptyset \in AC$ *(of course, also* $Afail \leq Afail$*);*
- $AC_1 \leq AC_2$ *iff* $AC_1 \subseteq AC_2$ *for all* $AC_1, AC_2 \in Con^{Fail} \setminus \{\bot, Afail\}$.

The minimal element in Con^{Fail} is $\bot$, the maximal element is $\wp(\wp(Var))$.

Definition 9.4.3 (Abstract least upper bound)
The abstract least upper bound lub is defined by the following table :

lub	$\bot$	$Afail$	AC_2
$\bot$	$\bot$	$Afail$	AC_2
$Afail$	$Afail$	$Afail$	$AC_2 \cup \{\emptyset\}$
AC_1	AC_1	$AC_1 \cup \{\emptyset\}$	$AC_1 \cup AC_2$

with $AC_1, AC_2 \in Con^{Fail} \setminus \{\bot, Afail\}$.

Definition 9.4.4 (Abstraction function)
The abstraction function α *is defined as follows :*

- $\alpha(\emptyset) = \bot$;
- $\alpha(\{false\}) = Afail$;
- $\alpha(\{C\})$ *with* $C \in SCons$ *is defined as in Section 5.2.3.1;*
- $\alpha(CS) = lub(\{\ \alpha(C) \mid C \in CS\ \})$.

Example 9.4.1
$\alpha(\{true\}) = \emptyset$.
$\alpha(\{true, false\}) = \{\emptyset\}$.
$\alpha(\{X = 3, X = 1\}) = \{\{X\}\}$ *whereas* $\alpha(\{X = 3, X = 1, false\}) = \{\emptyset, \{X\}\}$.
$\alpha(\{X = 3, X + Y = 5 \wedge Y > 1, false\}) = \{\emptyset, \{X\}, \{Y\}, \{X, Y\}\}$.

[9] The extension of the $\mathcal{M}$, $\mathcal{DF}$ and $\mathcal{DM}$ abstractions is similar.

[10] The way in which abstract conjunction induces possible failure (cf. below) is the reason for letting $\emptyset \in AC$ denote possible failure instead of having $Afail \in AC$; now *Afail* can only occur as an abstract constraint on itself, not as part of some other abstract constraint. This is in contrast with the concrete case where $\{false\}$ may occur on itself as well as being part of a set containing also satisfiable constraints, e.g. $\{false, X = 3 \wedge Y = 1, X + Y = 5\}$.

The concretisation function is determined by α and $\leq$ as before (Section 5.2.3.2).

The abstract operations are extended to deal with definite/possible failure.

Definition 9.4.5 (Abstract conjunction)
The abstract conjunction Λ *is defined by the following table :*

Λ	$\bot$	$Afail$	AC_2
$\bot$	$\bot$	$\bot$	$\bot$
$Afail$	$\bot$	$Afail$	$Afail$
AC_1	$\bot$	$Afail$	$join(AC_1, AC_2)$

with $join(AC_1, AC_2) = AC_1 \cup AC_2 \cup (AC_1 \oplus AC_2)$
and $AC_1 \oplus AC_2 = \{(A_1 \cup A_2) \setminus D \mid A_1 \in AC_1, A_2 \in AC_2, D \subseteq A_1 \sqcap A_2\}$.

Note that $\oplus$ may give rise to $\emptyset$ in the resulting abstraction. The presence of $\emptyset$ indicates *possible* failure. For example, consider $\{\{X, Y\}\} \Lambda \{\{X, Y\}\} = \{\emptyset, \{X\}, \{Y\}, \{X, Y\}\}$; the concrete conjunction corresponding to this abstract one could be $X + Y = 3 \wedge X - Y = 7$, in which case the resulting constraint is satisfiable, but it could also be $X + Y = 3 \wedge X + Y = 7$ which is unsatisfiable.
The safety condition still holds :
if $\alpha(CS_1) \leq AC_1$ and $\alpha(CS_1) \leq AC_2$, then $\alpha(CS_1 \wedge CS_2) \leq AC_1 \Lambda AC_2$.

Definition 9.4.6 (Abstract projection)
Let $V \subseteq Var$. *The abstract projection on* V *is defined as follows :*

- $\overline{\exists}_V \bot = \bot$;
- $\overline{\exists}_V Afail = Afail$;
- $\overline{\exists}_V AC = \{S \in AC \mid S \subseteq V\}$ *if* $AC \in Con^{Fail} \setminus \{\bot, Afail\}$.

Again, the safety condition still holds : if $\alpha(CS) \leq AC$, then $\alpha(\overline{\exists}_V CS) \leq \overline{\exists}_V AC$.

An extra primitive abstract operation $no\_fail(AC)$ is introduced to delete definite/possible failure information, such that it will not be propagated. It holds that $no\_fail(\alpha(CS)) = \alpha(CS \setminus \{false\})$.

Definition 9.4.7 (no_fail)

- $no\_fail(\bot) = \bot$;
- $no\_fail(Afail) = \bot$;
- $no\_fail(AC) = AC \setminus \{\emptyset\}$ *if* $AC \in Con^{Fail} \setminus \{\bot, Afail\}$.

The overall abstract interpretation procedure (cf. Algorithm 3.2.1) has to be adjusted at the following points :

- Instead of just testing for $AC_c = \bot$ at the start of the procedure, we now have : if $AC_c = \bot$ *or* $AC_c = Afail$, then $AC_c = \bot$.

- At the end of Case 2, we get : "Otherwise (if there is no clause for the goal L in the program, i.e. no clause whose head has the same predicate symbol p/k as L), then $AC_s = Afail$" (and not $\bot$ as in Algorithm 3.2.1).

The higher-level abstract operations (abstract interpretation of a constraint, procedure-entry, procedure-exit) are defined using abstract conjunction, abstract projection and *no_fail*. Note that they cannot be called with $AC_c = \bot$ or *Afail* (this is already captured by the abstract interpretation procedure, as mentioned above). At the time procedure-entry is called, the abstract interpretation procedure also implies that there exists a clause for the procedure. The abstract operations mimic the operations in the collecting semantics.

Definition 9.4.8 (Abstract interpretation of a constraint)
Let $C \in Cons$ and let $AC_c = (AC_c^o, AC_c^n)$ be the abstract call constraint of C.
If $\alpha(C) = Afail$, then $AC_s = Afail$. Otherwise, $AC_s = (AC_s^o, AC_s^n)$ is defined as :

- $AC_s^o = no\_fail(AC_c^o) \cup (no\_fail(AC_c^o) \ominus \alpha(C))$ *and*
- $AC_s^n = no\_fail(AC_c^n) \wedge \alpha(C)$.

It can be shown that[11] $no\_fail(AC_c^t) \wedge \alpha(C) = AC_s^t$. This is the basis for showing the safety of abstract interpretation of a constraint (similar as in Section 5.4).

Definition 9.4.9 (Procedure-entry)
Let (AC_c^o, AC_c^n) be the abstract call constraint of a procedure call $p(t_1, \ldots, t_k)$ and let $p(s_1, \ldots, s_k)$ be the head of a clause[12] used to resolve the call; $t_1, \ldots, t_k$ and $s_1, \ldots, s_k$ are general terms (at least for non-normalised programs). Then,

- $AC_{entry} = (\exists_{vars(t_1,\ldots,t_k)} no\_fail(AC_c^t), \emptyset)$;
- *If $\alpha(t_1 = s_1 \wedge \ldots \wedge t_k = s_k) = Afail$, then $AC_{in} = Afail$; otherwise,*
  $AC_{in} = (\exists_{vars(s_1,\ldots,s_k)}(AC_{entry}^o \wedge \alpha(t_1 = s_1 \wedge \ldots \wedge t_k = s_k)), \emptyset)$.

It can be shown that
- $\exists_{vars(t_1,\ldots,t_k)}(no\_fail(AC_c^t)) = AC_{entry}^t$;
- $\exists_{vars(s_1,\ldots,s_k)}(no\_fail(AC_c^t) \wedge \alpha(t_1 = s_1 \wedge \ldots \wedge t_k = s_k)) = AC_{in}^t$.

This is the basis for showing the safety of procedure-entry.

Definition 9.4.10 (Procedure-exit)
Let (AC_c^o, AC_c^n) be the abstract call constraint of a call $p(t_1, \ldots, t_k)$. Let the head of each clause Cl_i defining p/k be of the form $p(s_1, \ldots, s_k)$ $(1 \leq i \leq m)$. Let $AC_i = (AC_i^o, AC_i^n)$ be the abstract constraint at the end of Cl_i.
Then,

- $AC_{exit}^n = lub(\{\ldots, \exists_{vars(t_1,\ldots,t_k)}(AC_i^n \wedge \alpha(t_1 = s_1 \wedge \ldots \wedge t_k = s_k)), \ldots\})$
  (herein, let $AC_i^n = Afail$ or $\bot$ if $AC_i = Afail$ or $\bot$).
- *If $AC_{exit}^n = \bot$ or $Afail$, then $AC_s = \bot$; otherwise, the abstract success constraint (AC_s^o, AC_s^n) of the call is defined as*
  1. $AC_s^o = no\_fail(AC_c^o) \cup (no\_fail(AC_c^o) \ominus no\_fail(AC_{exit}^n))$
  2. $AC_s^n = no\_fail(AC_c^n) \wedge no\_fail(AC_{exit}^n)$

[11] By abuse of notation, if $AC = \bot$ (resp. *Afail*), assume that $AC^t = \bot$ (resp. *Afail*).
[12] The clause is assumed to be renamed apart from $vars(AC_c^t)$.

It can be shown that $AC_s^t = no\_fail(AC_c^t) \wedge no\_fail(AC_{exit}^n)$. This is the basis for showing the safety of procedure-exit.

The following proposition captures some important properties of the abstract success constraint AC_s of a procedure call.

Proposition 9.4.1
Let AC_s be the abstract success constraint of a procedure call p. Then

1. *$AC_s \neq Afail$*
2. *$AC_s = \bot$, given that $AC_c \notin \{\bot, Afail\}$, arises from* local *execution without interaction with the call environment. It indicates definite failure or unreachability due to an infinite recursive call, within each clause defining p; definite failure is caused by unsatisfiability of* local *constraints independent from the constraints projected at procedure-entry.*
3. *$\emptyset \in AC_s^t$ indicates possible failure due to conjunction of the local procedure constraints with the constraints projected at procedure-entry.*

PROOF

1. $AC_s \neq Afail$ since by definition of procedure-exit $AC_s^t = no\_fail(AC_c^t) \wedge no\_fail(AC_{exit})$ where the *no_fail* operation cannot yield *Afail*, and *Afail* can only be obtained via abstract conjunction with *Afail*.
2. By definition of procedure-exit, if $AC_c \notin \{\bot, Afail\}$, then $AC_s = \bot$ can only occur if $AC_{exit}^n = \bot$ or $AC_{exit}^n = Afail$. Hereby, $AC_{exit}^n = \bot$ or $AC_{exit}^n = Afail$ can only be caused by the detection of unreachability or definite failure within each clause defining the procedure. Definite failure during procedure analysis can only be detected due to unsatisfiability of *local* constraints; interference between local constraints and the ones projected at procedure-entry is computed via abstract conjunction, which can at most lead to the detection of *possible* failure (not *definite* failure).
3. Possible failure due to local constraints only, which may be reflected by the presence of $\emptyset$ in AC_{exit}^n, is not propagated into the success constraint AC_s of the procedure call since the application of *no_fail* in $AC_s^t = no\_fail(AC_c^t) \wedge no\_fail(AC_{exit}^n)$ does not propagate failure. However, possible failure during analysis of a procedure because of conjunction of the local constraints with the constraints projected at procedure-entry is rediscovered at procedure-exit. This is due to the recomputation at procedure-exit (cf. Section 5.4): the entire call constraint AC_c^t (including the part retained after procedure-entry) is joined with the local constraints represented by AC_{exit}^n; so if this conjunction gave rise to possible failure during local analysis of the procedure, it will also give rise to possible failure at procedure-exit. Hence, $\emptyset \in AC_s$ indicates possible unsatisfiability of the local procedure constraints with the call store. □

Finally, we want to make a remark on the treatment of future redundant constraints (cf. Section 9.2) in the context of a failure analysis. As mentioned in Section 9.2, a future-redundant constraint C does not have to be added to the constraint store. However, it is still necessary to check consistency of C with the store. If the analysis should cover failure information, the result of this consistency check should be reflected in the abstract success constraint AC_s of C. This means that AC_s cannot simply be equated to AC_c,

but must be computed as follows : first, usual abstract interpretation of C is performed, i.e. $AC_s^h = AC_c \wedge \alpha(C)$, where AC_c is the abstract call constraint of C; secondly, (1) if $AC_s^h = \bot$ (indicating unreachability) then $AC_s = \bot$, (2) if $AC_s^h = Afail$ (indicating definite failure) then $AC_s = Afail$, (3) if AC_s^h contains $\emptyset$ (indicating possible failure) then $AC_s = AC_c \cup \{\emptyset\}$, (4) otherwise $AC_s = AC_c$.

The definite/possible failure discovered by the above analysis does not include type clashes, such as a variable appearing in a numerical constraint that is afterwards bound to a non-numerical term. The failure information is therefore only useful for well-typed programs (note : most "real" programs are well-typed). Otherwise, the analysis has to be extended with type information, as explained in the next section.

9.5 Adding type information

The freeness analysis must be combined with type information in order to be useful for constraint specialisation. Types, in combination with modes, describe more precisely the set of possible values that a program variable can take at each program point (note : in the way we use it, the *type* of a variable is related to a specific program point; it does not necessarily hold over the entire program, in contrast with types supplied by a type declaration in a conventional programming language like Pascal). In the sequel, we still use the term "mode" to refer to a combination of mode and type information.
To perform constraint specialisation, the mode f (free), which can be derived from the abstract constraints in our freeness abstractions (Chapters 5, 6, 7 and 8), must be refined into $\mathbf{f}_u$, $\mathbf{f}_n$ and $\mathbf{f}_a$. A variable X has mode $\mathbf{f}_u$ if X is free and untyped, so X is not constrained to be a Herbrand or numerical variable (cf. Section 2.2 for the precise definition of *untyped*). A variable X has mode $\mathbf{f}_n$ if X is free and is of type numerical, so X occurs in a numerical constraint or occurs in a constraint $X = Y$ where Y is numerical. The mode $\mathbf{f}_a$ covers both $\mathbf{f}_u$ and $\mathbf{f}_n$; a variable with mode $\mathbf{f}_a$ is free and can be of type *untyped* or *numerical*. The mode $\mathbf{f}_u$ corresponds to the more restrictive notion of freeness used by Marriott and Søndergaard [83].

Example 9.5.1
Consider the constraint $C \equiv X = 3 \wedge X = T \wedge U + W = 2 \wedge Y = Z$. *The freeness analysis gives rise to the modes* $X : \mathbf{a}$, $T : \mathbf{a}$, $U : \mathbf{f}$, $W : \mathbf{f}$, $Y : \mathbf{f}$, $Z : \mathbf{f}$ (a *denotes mode any,* f *denotes mode free). After refinement :* $U : \mathbf{f}_n$, $W : \mathbf{f}_n$, $Y : \mathbf{f}_u$, $Z : \mathbf{f}_u$.

Only variables with mode $\mathbf{f}_u$ are useful for specialising constraints to assignments. A variable with mode $\mathbf{f}_n$ occurs in the numerical constraint store and hence the solver must be invoked when X is assigned a value. Although assigning a value to a free numerical variable does not override the satisfiability of the resulting store, it involves some extra solver operations to put the store back in solved form. For example, consider a variable X involved in a conjunction of equations C. If X is a parametric variable (i.e. X occurs only within the right-hand side of the equations in $sform(C)$), the right-hand side of every equation in which X occurs must be simplified if X is assigned a value n (more precisely n adds a contribution to the constant term of the equation). If X is a non-parametric variable, i.e. X occurs as the left-hand side of an equation $X = t$ in $sform(C)$, the equation $X = t$ turns into $n = t$ after assigning the value n to X. Then $n = t$ must be rewritten to

$Y = t'$ (choosing Y as new non-parametric variable) and also the rest of the equations in the store must be transformed (substituting Y by t' in every other equation which may in turn require further transformation). Whereas if X is a free untyped variable (mode $\mathbf{f_u}$), the constraint $X = b$, with b a ground numerical or non-numerical term, can be transformed into the assignment $X := b$, and requires no further transformation.

A minimal amount of type information is also useful to detect type clashes. A type clash is a special case of failure. Detection of type clashes yields more complete failure information, which combined with the results of the previous section, also allows to handle and optimise programs that are not well-typed.

A minimal type system for $CLP(H, N)$ is one including the types $\{$*Herbrand*, *numerical*, *anytype*$\}$ (*Herbrand* and *numerical* are defined in Section 2.2; *anytype* enables to assign a type to any program variable at any program point). This type system is closed under further constraining. It means that, given a variable of type *tp*, *tp* is still a valid type description when the variable is further constrained (in terms of [59], such a type system is *rigid*). A more extensive rigid type system is described by Gallagher and De Waal [40].

However, a type system of that kind does not provide the necessary information for constraint specialisation. For that purpose, the type system should be extended to $\{$*untyped*, *Herbrand*, *numerical*, *anytype*$\}$. Combination of type *untyped* with the freeness information (mode *free*) that can be derived from our freeness abstraction then allows to derive the mode $\mathbf{f_u}$, required for constraint specialisation[13]. The extended type system is no longer closed under further constraining (due to the description *untyped*). Consequently, information on possible variable dependencies is essential to maintain sufficient precision. Otherwise, any untyped variable becomes possibly typed (described by type *anytype*) after abstract conjunction.

Example 9.5.2
Let $C_1 \equiv A = B \wedge X = Y$. This constraint yields the following modes for its variables : $A : \mathbf{f_u}, B : \mathbf{f_u}, X : \mathbf{f_u}, Y : \mathbf{f_u}$; it establishes the dependencies $\{A, B\}$ and $\{X, Y\}$. Assume that $C_2 \equiv Y + Z = 3$ is joined to C_1. The correct modes then become : $A : \mathbf{f_u}, B : \mathbf{f_u}, X : \mathbf{f_n}, Y : \mathbf{f_n}, Z : \mathbf{f_n}$. An analysis not taking into account possible dependency information would necessarily derive $A : \mathbf{a_a}, B : \mathbf{a_a}, X : \mathbf{a_a}, Y : \mathbf{f_n}, Z : \mathbf{f_n}$, i.e. free and untyped variables become possibly free and typed after conjunction. A more accurate analysis (such as the one we propose in the sequel) will yield $A : \mathbf{f_u}, B : \mathbf{f_u}, X : \mathbf{f_a}, Y : \mathbf{f_n}, Z : \mathbf{f_n}$; so A and B remain free and untyped, X remains free but becomes possibly typed via a possible dependency with Y (which will be in $\alpha^{\mathcal{F}}(C_1)$).

Let $C_1 \equiv A = f(X) \wedge B = f(Y)$ and $C_2 \equiv A = B \wedge X + Z = 3$. The abstraction of C_1 yields the modes $A : \mathbf{a_h}, X : \mathbf{f_u}, B : \mathbf{a_h}, Y : \mathbf{f_u}$ and $\alpha^{\mathcal{F}}(C_1) = close(\{\{A\}, \{B\}, \{A, X\}, \{B, Y\}\})$. The abstraction of C_2 yields the modes $A : \mathbf{f_u}, B : \mathbf{f_u}, X : \mathbf{f_n}, Z : \mathbf{f_n}$ and

[13]The type system includes $\{$*untyped*, *Herbrand*, *numerical*, *anytype*$\}$; the mode system derivable from our freeness abstraction $\mathcal{F}$ includes the modes $\{$*any*, *free*$\}$. So the complete set of combinations is the following : $\{$*free* $-$ *anytype*($\mathbf{f_a}$), *free* $-$ *numerical*($\mathbf{f_n}$), *free* $-$ *untyped*($\mathbf{f_u}$), *any* $-$ *anytype*($\mathbf{a_a}$), *any* $-$ *numerical*($\mathbf{a_n}$), *any* $-$ *Herbrand*($\mathbf{a_h}$), *any* $-$ *untyped*($\mathbf{a_u}$)$\}$. The combination *free* $-$ *Herbrand* is impossible as a variable of type Herbrand is necessarily non-free. Although the combination *any* $-$ *untyped* is a valid description (*any* includes *free*), it will not occur in practice as it can always be replaced by a more precise description *free* $-$ *untyped* or *free* $-$ *numerical*.

$\alpha^{\mathcal{F}}(C_2) = close(\{\{A, B\}, \{X, Z\}\})$. Then joining the abstractions of C_1 and C_2 should yield that Y becomes possibly typed due to the entailed equation $X = Y$ established via the equation between the enclosing terms $f(X) = f(Y)$. Note that $\alpha^{\mathcal{F}}(C_1) \Lambda^{\mathcal{F}} \alpha^{\mathcal{F}}(C_2)$ will contain $\{X\}$ and $\{Y\}$, indicating that X and Y become possibly non-free and therefore also possibly typed (note: the imprecision in the freeness information is due to the fact that $\alpha^{\mathcal{F}}(C_1)$ describes e.g. also the constraint $A = f(X, 1) \wedge B = f(1, Y)$, in which case adding $A = B$ yields possible non-freeness of X and Y).

As our freeness abstraction already contains possible dependency information, it can quite easily be extended to derive the desired type information as well (we only consider the integration into the freeness analysis extended to detect definite/possible failure, defined in the previous section; integration into the other analyses of the previous chapters and sections can be performed in a similar way). The type information is added as an extra component to each abstract constraint.

Definition 9.5.1 (Abstract constraint)
An abstract constraint AC in the freeness analysis extended with type information is either $\perp$, Afail or is of the form $(Dep, Tp, TpClash)$ where

- *Dep represents the possible dependency information concerning the variables in the domain of AC (i.e. the set of variables of the clause or goal to which AC is attached), cf. freeness analysis in Section 5;*
- *$Tp = \{(X, tpX) \mid X$ in the domain of $AC, tpX \in \{untyped, Herbrand, numerical, anytype\}\}$ contains the type tpX of each variable X in the domain of AC;*
- *$TpClash \in \{yes, no\}$ is a flag that indicates whether the constraints abstracted by AC* possibly *cause a type clash.*

The abstract constraint $\perp$ denotes unreachability, *Afail* denotes *definite* failure due to a value clash or type clash, $\emptyset \in Dep$ denotes *possible* failure due to a value clash and $TpClash = yes$ denotes *possible* failure due to a type clash.

The abstraction of a constraint C is straightforward. If C is unsatisfiable, due to either a value or a type clash, the abstraction of C is *Afail*. Otherwise, the abstraction $(Dep, Tp, TpClash)$ is obtained as follows :

- The dependency information is derived using the abstraction function of Section 5.2.3.1.
- The type information extracted from C has been defined in Section 2.2 : for each $X \in vars(C)$, the Tp component contains
  - $(X, Herbrand)$ if $sform(C)$ contains $X = t$ with $t \notin T(\Sigma_N, Var)$;
  - $(X, numerical)$ if X occurs in a primitive numerical constraint in C or if X is bound to a numerical variable Y via $X = Y$ or $Y = X$ in $sform(unif^*(C))$;
  - $(X, untyped)$ otherwise.
- $TpClash = no$ (note : when abstracting a constraint C, a *definite* type clash may be detected resulting in the abstraction *Afail*; only abstract conjunction may cause the detection of a *possible* type clash).

Example 9.5.3
Let $C \equiv X = f(Y,Z) \wedge Z = T \wedge Y = 1$. *Then* $\alpha(C) = (close(\{\{X\},\{Y\},\{X,Z\},\{Z,T\},\{X,T\}\}),\ \{(X : Herbrand),(Y : numerical),(Z : untyped),(T : untyped)\},\ no)$.
Let $C \equiv X = Y \wedge Y + Z = 5$. *Then* $\alpha(C) = (close(\{\{X,Y\},\{Y,Z\},\{X,Z\}\}),\ \{(X : numerical),(Y : numerical),(Z : numerical)\},\ no)$.

As before, the abstraction of a set of constraints is based on the abstraction of a single constraint and the abstract least upper bound (defined below).
The order relation on abstract constraints is based on the order relation on their components.

Definition 9.5.2 (Abstract computational order)
1. *for the order involving the special abstract constraints* $\bot$ *and Afail we refer to the previous section;*
2. $(Dep_1, Tp_1, TpClash_1) \leq (Dep_2, Tp_2, TpClash_2)$ *(the abstract constraints are assumed to have the same domain) iff*
   - $Dep_1 \subseteq Dep_2$;
   - *for each* X *with* $(X, tpX_1) \in Tp_1$ *and* $(X, tpX_2) \in Tp_2$: $tpX_1 \leq tpX_2$, *where* $tp_1 \leq tp_2$ *iff* $tp_1 \equiv tp_2$ *or* $tp_2 = anytype$;
   - $TpClash_1 \leq TpClash_2$ *where* $yes \leq no$, $yes \leq yes$ *and* $no \leq no$.

Definition 9.5.3 (Abstract least upper bound)
1. $lub(\bot, AC) = lub(AC, \bot) = AC$ *for each* AC;
2. $lub(Afail, Afail) = Afail$ *and*
   $lub(Afail, (Dep, Tp, TpClash)) = lub((Dep, Tp, TpClash), Afail) = (Dep \cup \{\emptyset\}, Tp, yes)$
3. $lub((Dep_1, Tp_1, TpClash_1), (Dep_2, Tp_2, TpClash_2)) = (Dep, Tp, TpClash)$ *(the abstract constraints are assumed to have the same domain) where*
   - $Dep = Dep_1 \cup Dep_2$;
   - $Tp = \{(X, tpX) \mid tpX = lub(tpX_1, tpX_2), (X, tpX_1) \in Tp_1, (X, tpX_2) \in Tp_2\}$ *where* $lub(tp_1, tp_2) = tp_1$ *if* $tp_1 \equiv tp_2$ *and* $lub(tp_1, tp_2) = anytype$ *otherwise;*
   - $TpClash = no$ *if* $TpClash_1 = TpClash_2 = no$, *otherwise* $TpClash = yes$.

The abstract projection of an abstract constraint on a set of variables $V \subseteq Var$ is defined as follows.

Definition 9.5.4 (Abstract projection)
1. *for the abstract projection of* $\bot$ *and Afail on* V *we refer to the previous section;*
2. $\exists_V(Dep, Tp, TpClash) = (Dep_{proj}, Tp_{proj}, TpClash)$ *with*
   - $Dep_{proj} = \exists_V^{\mathcal{F}} Dep$;
   - $Tp_{proj} = \exists_V^{\mathcal{T}} Tp = \{(X, tpX) \in Tp \mid X \in V\}$.

The operation *no_fail*, which deletes failure information such that it will not be propagated, is defined as follows.

Definition 9.5.5 (no_fail)
1. *for the result of no_fail on* $\bot$ *and Afail we refer to the previous section;*
2. $no\_fail(Dep, Tp, TpClash) = (Dep \setminus \{\emptyset\}, no)$.

The abstract conjunction of two abstract constraints is somewhat more complex. For a conjunction involving $\bot$ or *Afail* we refer to the previous section. The abstract conjunction $(Dep_1, Tp_1, TpClash_1) \wedge (Dep_2, Tp_2, TpClash_2)$ is defined by the algorithm below.

Algorithm 9.5.1 (Abstract conjunction)
Let $AC_1 = (Dep_1, Tp_1, TpClash_1)$ and $AC_2 = (Dep_2, Tp_2, TpClash_2)$ be two abstract constraints having the same domain[14].
Then $(Dep_1, Tp_1, TpClash_1) \wedge (Dep_2, Tp_2, TpClash_2) = (Dep, Tp, TpClash)$ where $Dep = Dep_1 \wedge^{\mathcal{F}} Dep_2$ using the abstract conjunction operation defined in the previous section and Tp and $TpClash$ are computed through the following steps :

1. *$TpClash$ is initialised to "no"; it may be overridden by one of the following steps.*
2. *The type components are joined, without yet taking into account the possible variable dependencies. The conjunction of types, denoted $\wedge^T$, is defined as follows :*

$\wedge^T$	*untyped*	*Herbrand*	*numerical*	*anytype*
untyped	*untyped*	*Herbrand*	*numerical*	*anytype*
Herbrand	*Herbrand*	*Herbrand*$^+$	$\bot$	*Herbrand*$^+$
numerical	*numerical*	$\bot$	*numerical*	*numerical*$^+$
anytype	*anytype*	*Herbrand*$^+$	*numerical*$^+$	*anytype*$^+$

   The superscript "+" in the table indicates a possible type clash (note : the conjunction of Herbrand with Herbrand may result in a possible type clash for one of the components of the Herbrand terms, e.g. $X = f(1)$ and $X = f(a)$). A provisional value of Tp is computed by joining Tp_1 and Tp_2 using $\wedge^T$: $Tp = \{(X, tpX) \mid (X, tpX_1) \in Tp_1, (X, tpX_2) \in Tp_2, tpX = tpX_1 \wedge^T tpX_2\}$. If at least one of the $tpX = \bot$, then the resulting abstract constraint is $\bot$ instead of $(Dep, Tp, TpClash)$ and the subsequent steps in the algorithm can be discarded. If for at least one variable a possible type clash is detected, corresponding to one of the "+" cases in the table above, then $TpClash := yes$.

3. *The Dep, Dep_1 and Dep_2 information is used to adjust the type information Tp derived in the previous step. More precisely, the description* untyped *in Tp for a variable X may be overruled due to a possible (entailed) equation between X and a variable that is not* untyped. *At the same time, possible type clashes due to the (entailed) equations will be discovered. There are two ways in which the type of a variable $X \in \{Z \mid tpZ = untyped\}$ can be influenced and due to which a possible type clash can arise :*

   a. *If X is possibly non-free after joining Dep_1 and Dep_2, i.e. $\{X\} \in Dep$, then X is also possibly typed, so $tpX := anytype$ (this follows immediately from the definitions of possible non-freeness and the type definitions). Note that this also covers those cases in which X possibly becomes typed because it is a component of a Herbrand term which gets bound to another Herbrand term by combining AC_1 and AC_2 (i.e. there is a possible equation between X and a term t, that is not a variable, via an entailed equation between two enclosing terms, cf. Example 9.5.2).*
   If $\emptyset \in Dep$, this indicates a possible value clash and hence also a possible type clash, so $TpClash := yes$.

[14] If the domains of the abstract constraints differ, their Tp components can be extended to include the same variables by adding a pair $(X, untyped)$ for each variable that is missing.

b. Even if X is still free in Dep (i.e. $\{X\} \notin Dep$), X may possibly become typed due to a possible equation $X = Y$ with a possibly typed variable Y. We only have to consider "flat" equations (i.e. equations that are not entailed by going via an entailed equation between compound Herbrand terms); the other cases are already covered by step a. A dependency $\{X, Y\}$ may correspond to an equation $X = Y$ if X and Y are possibly *untyped, so their type is either* untyped *or* anytype. *This gives rise to the following steps :*

- Construct the equivalence classes $S_1, \ldots, S_n$ $(n \geq 0)$ of all variables that are possibly equated together via $Dep_1 \cup Dep_2$ [15]*, as follows :*
let $H = \{\{X, Y\} \in Dep_j \mid j \in \{1,2\},\ tpX_j = untyped \mid anytype, tpY_j = untyped \mid anytype\}$[16]*;*
interpret $\{X, Y\} \in H$ as "there is a possible equality relation between X and Y" and construct the equivalence classes $S_1, \ldots, S_n$ of the variables in H using this equivalence relation.

- for each S_i :
let $types(S_i) = \{tpX \mid X \in S_i\}$;
if $types(S_i) = \{untyped\}$
then nothing changes
else compute the conjunction of all types in $types(S_i)$ using $\wedge^T$ defined above;
if the result of the conjunction is $\bot$ or contains the superscript "+"
then $TpClash := yes$;
for each $X \in S_i$ with $tpX = untyped$: $tpX := anytype$

A more precise abstract conjunction operation can be obtained if one of the conjuncts (assume AC_2) is the abstraction of a single constraint. In that case, the dependencies are *definite* dependencies, which means that there *certainly* (not *possibly*) exist primitive constraints that establishes them. To improve precision, the following extra step should be performed before step 3.a in the algorithm above :

- Construct the equivalence classes $S_1^2, \ldots, S_n^2$ $(n \geq 0)$ of all variables equated together via Dep_2 :
let $H_2 = \{\{X, Y\} \in Dep_2 \mid tpX_2 = untyped, tpY_2 = untyped\}$[17];
interpret $\{X, Y\} \in H_2$ as "there is a sure equality relation between X and Y" and construct the equivalence classes $S_1^2, \ldots, S_n^2$ of the variables in H_2 using this equivalence relation.
- for each S_i^2 :

[15] Considering Dep instead of $Dep_1 \cup Dep2$ would not yield as precise results, as it is impossible to find out which pairs $\{X, Y\}$ in Dep correspond to equations of the form $X = Y$. The reason is that the variable types after combining Dep_1 and Dep_2 to Dep, which are described by Tp after step a, are already imprecise. Consider $C_1 \equiv X = f(Y)$ and $C_2 \equiv X = Z$. Then $Dep = close(\{\{X\}, \{Z\}, \{X, Y\}, \{Y, Z\}\})$ and, after step a, $Tp = \{(X, Herbrand), (Y, untyped), (Z, anytype)\}$; so $\{Y, Z\}$ could be interpreted as an equation of the form $Y = Z$ such that Y would become of type *anytype*. However, this is not precise as the concrete conjunction $C_1 \wedge C_2$ leaves Y *untyped*. Using the algorithm based on $Dep_1 \cup Dep_2$, the more precise mode *untyped* can be derived for Y.
Another thing to note is that all (entailed) dependencies, also the ones entailed by combining Dep_1 with Dep_2, are represented exhaustively in Dep, whereas when considering $Dep_1 \cup Dep_2$ one still has to compute the equivalence classes of variables linked together due to the combination.

[16] The types considered are the original types in AC_j, not the ones obtained after step 2 or step 3a.

[17] tpX_2 (or tpY_2) cannot be *anytype* in the abstraction of a single constraint.

let $types(S_i^2) = \{tpX \mid X \in S_i^2\}$;
if $types(S_i^2) = \{untyped\}$
then nothing changes
else compute the conjunction of all types in $types(S_i^2)$ using Λ^T defined above;
if the result of the type conjunction is $\bot$
then the constraint resulting from the abstract conjunction is $\bot$
(go to end of the algorithm)
else if the result of the type conjunction contains the superscript "+"
then $TpClash := yes$;
for each $X \in S_i^2$:
if $types(S_i^2)$ contains t with $t \in \{Herbrand, numerical\}$[18]
then $tpX := t$
else $tpX := anytype$

Example 9.5.4
Let $C_1 \equiv X = f(Y)$ and $C_2 \equiv Y = 3$; $\alpha(C_1) = (\{\{X\},\{X,Y\}\}, \{(X, Herbrand),(Y, untyped)\}, no)$ and $\alpha(C_2) = (\{\{Y\}\}, (Y, numerical), no)$. Before applying abstract conjunction, the type component of $\alpha(C_2)$ is extended with $(X, untyped)$. Then $\alpha(C_1) \Lambda \alpha(C_2) = (\{\{X\},\{X,Y\},\{Y\}\}, \{(X, Herbrand),(Y, numerical)\}, no)$.

Let $C_1 \equiv X = Y \wedge Z = 3$ and $C_2 \equiv Y = Z$; $\alpha(C_1) = (\{\{X,Y\},\{Z\},\{X,Y,Z\}\}, \{(X, untyped),(Y, untyped),(Z, numerical)\}, no)$ and $\alpha(C_2) = (\{\{Y,Z\}\}, \{(Y, untyped),(Z, untyped)\}, no)$. Before applying abstract conjunction, the type component of $\alpha(C_2)$ is extended with $(X, untyped)$. Then $\alpha(C_1) \Lambda \alpha(C_2) = (close(\{\{X\},\{Y\},\{Z\}\}), \{(X, anytype),(Y, anytype),(Z, numerical)\}, no)$. The imprecise type information for X and Y is due to the fact that the dependencies arising from $X = Y$ and $Y = Z$ are considered as possible dependencies when performing abstract conjunction. Now assume that $\alpha(C_1)$ is the store at some point and the abstraction of $Y = Z$ should be added to this abstract store; then the dependency $\{Y, Z\}$ is considered as a definite dependency whereas $\{X, Y\}$ is already considered as a possible dependency. The improved abstract conjunction algorithm then yields : $\alpha(C_1) \Lambda \alpha(C_2) = (close(\{\{X\},\{Y\},\{Z\}\}), \{(X, anytype),(Y, numerical),(Z, numerical)\}, no)$. So more precise type information is obtained for Y but not for X.

The following example involves sets of constraints. Let $CS_1 \equiv \{X = 1, X = f(Y)\}$ and $CS_2 \equiv \{X + Z = 0\}$; $\alpha(CS_1) = (\{\{X\},\{X,Y\}\}, \{(X, anytype),(Y, untyped)\}, no)$ and $\alpha(CS_2) = (\{\{X,Z\}\}, \{(X, numerical),(Z, numerical)\}, no)$. Before applying abstract conjunction, the type component of $\alpha(CS_1)$ is extended with $(Z, untyped)$ and the type component of $\alpha(CS_2)$ is extended with $(Y, untyped)$. Then $\alpha(CS_1) \Lambda \alpha(CS_2) = (close(\{\{X\},\{X,Y\},\{Z\},\{Y,Z\}\}), \{(X, anytype),(Y, anytype),(Z, numerical)\}, yes)$. A possible type clash is detected (whereas the dependency information does not give rise to possible failure).

In order to obtain precise results at procedure-exit, the Dep component of the abstract constraints has to be split into Dep^o and Dep^n, as for the freeness abstraction. The abstract

[18] $types(S_i^2)$ cannot contain both *Herbrand* and *numerical*, because then the algorithm would have stopped with $\bot$ as result (due to a definite type clash).

interpretation of a constraint, procedure-entry and procedure-exit are defined in a similar way as for the freeness abstraction, using the abstract conjunction and abstract projection operations defined above. Some points are worth mentioning :

- When the Tp components in the abstract conjunction operation defined above use the Dep information, Dep should be read as $Dep^t = Dep^o \cup Dep^n$,
- Procedure-entry computes the abstract constraint AC^j_{in} at the beginning of each clause Cl^j that can be used to resolve a procedure call. Compared to the procedure-entry operation of the freeness analysis, an extra step is needed here to initialise the local clause variables (these are the variables occurring only in the body and not in the head of Cl^j) : for each local variable X in Cl^j, $(X, untyped)$ is added to the type component of AC^j_{in}.
- The abstract interpretation of a constraint C can use the improved abstract conjunction operation defined above, as one of the involved conjuncts is the abstraction of C.

Instead of integrating type inference into the freeness analysis, another way to obtain the necessary type information is to produce it via an independent analysis and use the type-annotated programs as input programs for the freeness analysis. However, as the accurate derivation of type *untyped* requires possible dependency information anyway, it seems more natural and more efficient to follow the approach presented here.

9.6 Analysis of other constraint domains

So far, attention was restricted to the analysis of CLP(H,N) languages. However, the freeness analyses presented in previous chapters can be extended to incorporate other constraint domains. The key point is to define the notion of freeness for the considered constraint domain, and the notion of variable dependencies to be used for non-freeness propagation. As an example we extend the analysed language with the domain of PrologIII tuples (also called the domain of strings in [55]).

9.6.1 The domain of PrologIII tuples

Terms in the PrologIII tuple domain [100] are defined by the following syntax :

term::= term2 | term2 "." term
term2::= variable | "⟨⟩" | "⟨" sequence_of_terms "⟩"
sequence_of_terms::= term | term "," sequence_of_terms

The empty tuple is denoted by the constant ⟨⟩; "." denotes tuple concatenation and "⟨" *sequence_of_terms* "⟩" denotes a sequence of tuple elements. A sequence $\langle X_1, \ldots, X_n \rangle$ $(n \geq 1)$ is equivalent to $\langle X_1 \rangle . \ldots . \langle X_n \rangle$.

Example 9.6.1
$\langle X, Y, Z \rangle$ denotes the tuple with elements X, Y and Z. $\langle X, Y \rangle . Z$ denotes the tuple with X and Y as its first two elements and Z representing the rest of the tuple. $X . \langle \langle A \rangle, Y \rangle . Z$ illustrates the possible nesting of tuples; it denotes the tuple which is the concatenation of the tuple X, the sequence of elements $\langle A \rangle$ (which is itself a tuple containing the single element A) and Y, and the tuple Z.

The above syntax corresponds to normalised programs in which tuple terms are strictly separated from other terms; a link with other domains (e.g. the Herbrand and the numerical domain) can be made by binding a tuple element (which is a variable) to a term in those domains. Of course, the PrologIII system provides a more practical, non-normalised syntax:

term::= term2 | term2 "." term
term2::= variable | "⟨⟩" | "⟨" sequence_of_terms "⟩"
sequence_of_terms::= term3 | term3 "," sequence_of_terms
term3::= Herbrand term | numerical term | term

The primitive constraints on tuples are equations and disequations between two tuples. An equation (or disequation) of the form $t_1 = t_2$ ($t_1 \neq t_2$) is a passive constraint (i.e. is delayed) if t_1 and/or t_2 is a concatenation in which the size of the leftmost operand is not known.
The size constraint provides a link between the tuple and the numerical domain (another link besides the fact that tuples may contain numerical terms as elements in non-normalised programs); it expresses a relation between a tuple and its size (i.e. its number of elements). The size constraint is of the form *tuple_term* :: *numerical_term* (in normalised form : X :: N where X and N are variables, X represents a tuple and N represents a number). It automatically implies the constraint *numerical_term* ≥ 0. A size constraint is delayed if neither the size of the first argument nor the value of the second argument are known.

We say that a tuple is of *restricted size* if either it is empty (size = 0), or it contains at least one element (size ≥ 1, so it is not just a variable, or a concatenation of variables and empty ⟨⟩ parts (including at least one variable)), or it contains less than a fixed number of elements (size $\leq n$ with n a number). Note : the fact that its size is automatically restricted to be positive (≥ 0) does not yet imply that it is of restricted size.

The CLP language including the PrologIII tuple domain is denoted by CLP(H,N,T). So, a constraint in CLP(H,N,T) is a conjunction of primitive constraints including equations and disequations of H-terms, equations, disequations and inequalities of N-terms, equations and disequations of T-terms and size constraints between T-terms and N-terms.

9.6.2 Freeness and dependencies in CLP(H,N,T)

The definition of freeness has to be extended towards CLP(H,N,T).

Definition 9.6.1 (freeness)
Let C be a constraint in CLP(H,N,T). A variable X is free with respect to C iff

1. *C does not entail a constraint $X \diamond t$ where $\diamond \in \{=, \neq\}$ and t is a non-variable Herbrand term or t is a tuple term of restricted size, and C does not entail a constraint $X \diamond n$ where $\diamond \in \{=, \neq, >, \geq\}$ and n is a number;*
2. *C does not entail a pair of constraints $X :: t$ and ($t = 0$, or $t \geq 1$, or $t \leq n$) where n is a number and t is a numerical term.*

We say that C constrains X if X is non-free in C.

Definition 9.6.2 (dependency)
A constraint C establishes a dependency between a set of variables $\{X_1, \ldots, X_n\}$ with $n \geq 2$ iff (further) constraining all but one of the variables may (further) constrain the remaining variable.

Further constraining X is done by joining to C (1) a constraint $X \diamond t$ where $\diamond \in \{=, \neq\}$ and t is a non-variable Herbrand term or a tuple term of restricted size, or a constraint $X \diamond n$ where $\diamond \in \{=, \neq, >, \geq\}$ and n is a number, or (2) a pair of constraints $X :: t$ and ($t = 0$ or $t \geq 1$ or $t \leq n$) where n is a number and t is a numerical term.

Example 9.6.2
Let $C \equiv X = \langle A \rangle . Y$. Then X is non-free, whereas A and Y are free. Further constraining X may further constrain A and vice versa, the same holds for X and Y; so C establishes the dependencies $\{X, A\}$ and $\{X, Y\}$ (note : only the minimal dependencies are mentioned).
Let $C \equiv X = Y . Z$. All variables are free and the established dependencies are $\{X, Y\}$ and $\{X, Z\}$.
Let $C \equiv X::N \wedge T::3 \wedge X = Y . Z$. Then X, Y and Z are free, T is non-free (its size is constrained to 3) and N is non-free (since C entails $N \geq 0$). The minimal dependencies established by C are $\{X, N\}$, $\{X, Y\}$, $\{X, Z\}$, $\{Y, N\}$ and $\{Z, N\}$; for example, for $\{Y, N\}$, further constraining N, e.g. by adding $N \leq 4$, further constrains Y (more precisely, the size of Y is restricted to be ≤ 4); vice versa, further constraining Y, e.g. by adding $Y = \langle A, B \rangle . D$, further constrains N ($N \geq 2$).

9.6.3 Abstraction of a constraint

The aim is to extend the original freeness abstraction (Chapter 5) to deal with CLP(H,N,T) programs. Having defined the notions *freeness* and *dependency*, the general definition of the abstraction function (Section 5.2.3.1) can be applied to compute the abstraction of a constraint C in CLP(H,N,T). However, this definition is not very practical. We now specialise it towards the abstraction of tuple constraints.

Assume that programs are normalised, so the tuple part of a constraint C, denoted $tup(C)$, can be separated from the numerical part, $num(C)$, and the Herbrand part, $unif(C)$. We define $tup(C)$ as the conjunction of primitive equations and disequations between tuple terms and of size constraints of the form $X :: N$, where X and N are variables; the elements of tuples are variables.

The solved form $sform(tup(C))$ where $tup(C)$ contains only equations and size constraints is obtained by subsequently applying one of the following rewrite rules in the given order, until no further transformation is possible (note : $\langle X_1, \ldots, X_n \rangle$ is assumed to be replaced by the equivalent form $\langle X_1 \rangle . \ldots . \langle X_n \rangle$) :

1. $\langle X \rangle . t_1 = \langle Y \rangle . t_2 \wedge C_{rest} \rightarrow X = Y \wedge t_1 = t_2 \wedge C_{rest}$;
2. $\langle \rangle . t_1 = t_2 \wedge C_{rest} \rightarrow t_1 = t_2 \wedge C_{rest}$;
3. $t = \langle \rangle \wedge C_{rest}$ where t is not a variable and $t \not\equiv \langle \rangle \rightarrow \langle \rangle = t \wedge C_{rest}$;
4. $t = X \wedge C_{rest}$ where t is not a variable $\rightarrow X = t \wedge C_{rest}$;
5. $\langle \rangle = t \wedge C_{rest}$ where t is not a variable $\rightarrow$ if $t \equiv \langle \rangle$ then C_{rest} else *false*;
6. $X = X \wedge C_{rest} \rightarrow C_{rest}$;

7. $X = t \wedge C_{rest}$ where $t \neq X$ and X has another occurrence in C_{rest}
   $\rightarrow$ if $X \in vars(t)$ then *false* else $X = t \wedge C_{rest}[X/t]$, i.e. X is replaced by t in every equation of C_{rest};
8. $X :: N \wedge X :: M \wedge C_{rest} \rightarrow X :: N \wedge C_{rest}$ and $N = M$ is added to $num(C)$.

A primitive constraint in $sform(tup(C))$ is of one of the following forms (for each form, the abstraction is specified) :

- $X = t$ where t is a tuple term (special case : $X = Y$)
  If t is of restricted size, then X is non-free (represented by $\{X\}$ in the abstraction). The constraint also establishes a dependency between X and each variable in t. So $\alpha^{\mathcal{F}}(X = t) = close(W)$ with $W = \{\{X\} \mid \text{if } t \text{ is of restricted size}\} \cup \{\{X, Y\} \mid Y \in vars(t)\}$.
- $X \,.\, t_1 = t_2$ where t_1 and t_2 are tuple terms
  For this constraint, a safe but rough abstraction is computed. If t_1 and/or t_2 is of restricted size, then $\alpha^{\mathcal{F}}(X \,.\, t_1 = t_2) = close(W)$ with $W = \{\{Y\} \mid Y \in \{X\} \cup vars(t_1, t_2)\}$.
  Otherwise $\alpha^{\mathcal{F}}(X \,.\, t_1 = t_2) = close(W)$ with $W = \{\{Y, Z\} \mid Y, Z \in \{X\} \cup vars(t_1, t_2)\}$.
- $X :: N$
  This constraint establishes a dependency $\{X, N\}$ and constrains N (represented by $\{N\}$ in the abstraction) since it entails $N \geq 0$. So $\alpha^{\mathcal{F}}(X :: N) = \{\{X, N\}, \{N\}\}$.

Disequations between tuple terms are abstracted in the same way as equations (as though $\neq$ is replaced by $=$), yielding a safe but imprecise abstraction.

The abstraction of a tuple constraint, i.e. a conjunction of primitive equations and disequations and size constraints, is defined by using the abstract conjunction operation $\Lambda^{\mathcal{F}}$ (Definition 5.3.1) of the freeness analysis.

Definition 9.6.3 (Abstraction of a tuple constraint)
Let C be a tuple constraint. Let $sform(C) = \{c_1, \ldots, c_n\}$ with $n \geq 1$. Then $\alpha^{\mathcal{F}}(C) = \alpha^{\mathcal{F}}(c_1)\, \Lambda^{\mathcal{F}} \ldots \Lambda^{\mathcal{F}} \alpha^{\mathcal{F}}(c_n)$. (note : if $sform(C) = false$ then $\alpha^{\mathcal{F}}(C) = \bot$).

Example 9.6.3
Let $C_1 \equiv X = Y \,.\, Z \wedge X ::N$. Then $\alpha^{\mathcal{F}}(C_1) = close(W_1)$ with $W_1 = \{\{X, Y\}, \{X, Z\}, \{N\}, \{X, N\}, \{Y, N\}, \{Z, N\}\}$. So all variables are free except N.
Let $C_2 \equiv X = (Y) \,.\, Z \wedge X ::N$. Then $W_2 = W_1 \cup \{\{X\}\}$ and $\alpha^{\mathcal{F}}(C_1) = close(W_2)$. So all variables are free except X and N.
Let $C_3 \equiv (A) \,.\, X = (B) \,.\, Y$. Then $\alpha^{\mathcal{F}}(C_3) = close(\{\{A, B\}, \{X, Y\}\})$. All variables are still free. There is a dependency between the tuple elements A and B, and between the tuples X and Y.

The abstraction of a general constraint C (including Herbrand, numerical and tuple constraints) is obtained by (1) computing the abstraction of the tuple part as defined above, (2) computing the abstraction of the Herbrand and numerical part (Definition 5.2.13)[19], where the numerical part is possibly augmented with constraints of the form $M = N$ resulting from the tuple part (case 9 in the solved form algorithm), and (3) joining the two abstractions using $\Lambda^{\mathcal{F}}$.

[19] Instead of considering the Herbrand and the numerical part as a whole, one could also compute the abstraction of the Herbrand part and the abstraction of the numerical part separately, and join them using $\Lambda^{\mathcal{F}}$. This yields the same abstraction as the one obtained by Definition 5.2.13.

The (primitive) abstract operations are the same as for the freeness analysis (Chapter 5).

A final remark concerns the treatment of the passive constraints. As for non-linear constraints, we again chose not to model delay and wake-up. Passive constraints are treated as though they become active at once.

Chapter 10

Implementation

We have built a prototype for the analyses described in Chapters 5, 6, 7 and 8, embedded in the abstract interpretation system PLAI [94, 96]. The prototype also includes some of the extensions mentioned in Chapter 9. The results in Appendix A are computed with this prototype. In this chapter we first discuss the impact of the implementation language. Secondly, the relevant properties of the PLAI system are described. The third section contains various issues concerning the implementation of the freeness abstraction (choice of data-structures, efficient algorithms for the abstract operations, etc.). It also mentions what features are actually incorporated in the current prototype. Finally, we point out how the combination of the definiteness with the $\mathcal{DF}$ and $\mathcal{DM}$ analysers (these are the freeness analysers exploiting definiteness information) is realised.

10.1 Choice of implementation language

The basic choice concerning the implementation language is between a logic or an imperative language. A logic language supports rapid prototyping which is interesting while experimenting with different abstract domains or changes in the implementation of one domain. This was our main motivation to select an abstract interpretation system written in a logic language, as we wanted to try out and evaluate different freeness abstractions. The different analysers have been obtained with a very small effort. The well-known advantage of using an imperative language, for example *C*, is a better performance both with respect to space and time. When the analysis system is to be integrated in a commercial (C)LP compiler, the performance is a crucial factor. For the original freeness abstraction, an alternative implementation using the abstract interpretation system GAIA [70, 39], written in C, has been developed by Simoens. This work also includes experiments with alternative C data-structures for the abstract constraints. Performance results of the C analyser and a comparison with the ones of the Prolog analyser are reported and evaluated in [61]; this paper also contains a more general comparison of two abstract interpretation systems (GAIA and PLAI), one written in C and the other written in Prolog.

When choosing a logic-based language, the next option is between a pure logic programming language or a constraint logic programming language. Especially when developing an analyser for a *constraint* logic programming language, it seems obvious to implement the analyser in the same language. One advantage is that the transformation of program

constraints to solved form before abstracting them can be done automatically using the constraint solver of the CLP system. Using an LP language requires the additional implementation of the solved form algorithm, or the avoidance of the algorithm by restricting the abstraction to *primitive* constraints (cf. further); however, the latter implies a loss of precision, since the abstract conjunction operation is then used to join the abstractions of the primitive constraints. Unfortunately, there are currently two reasons which determined our choice of a pure logic programming language as the implementation tool. First of all, to our knowledge, there does not yet exist a generic and efficient abstract interpretation system written in a CLP language. However, there is such a system written in Prolog at our disposal, namely the system PLAI [94, 96]. This system has already proven its usefulness in the implementation of other abstract domains for Prolog [95]. It also incorporates several optimisations of the abstract interpretation algorithm described by Bruynooghe in [6], which considerably increase the efficiency. A second reason for not implementing our analysers in a CLP language is the following. Developing an abstract interpretation system in CLP involves meta-programming. Although the CLP(R) system offers the necessary facilities for this, CLP(R) only covers the Herbrand domain and the domain of real numbers. These constraint domains are the main ones studied in this thesis, but we also consider extension to other constraint domains such as the domain of PrologIII tuples. Therefore, we did not want to restrict ourselves to CLP(R) as implementation language. Another language choice could be PrologIII. The difficulty here is that the PrologIII system available when we implemented our analysers, being PrologIII version 1.3, does not offer the necessary meta-programming features. This should be overcome by the new version, version 1.4, of the PrologIII system. Another issue is that the current CLP systems are still in an experimental phase, hence they are likely to be less efficient than the by now well established Prolog systems.

10.2 Abstract interpretation system PLAI

Our analysers are implemented within the abstract interpretation system PLAI [94, 96]. This system is written in SICStus Prolog. The abstract interpretation algorithm is a practical adaptation of the one designed by Bruynooghe [6]. It incorporates several optimisations which considerably improve the efficiency. These optimisations are summarised below (we use the same naming conventions for abstract constraints as in Section 3.2.3.2). A detailed description of the algorithm can be found in [96, 94].

- The algorithm does not explicitly build the AND-OR-graph. Instead, the information is implicitly contained in a memo table. During analysis, the memo table has an entry for each subgoal with distinct syntactic form (up to renaming of variables) and distinct abstract entry constraint, that has occurred so far. Besides the subgoal and entry constraint, it stores the – possibly incomplete – abstract exit constraint associated with the entry constraint. The exit information is incomplete if the predicate of the subgoal is recursive and the fixpoint for its exit constraint is not yet reached (e.g. the exit constraint is obtained using only the results of some clauses defining the predicate, cf. Section 3.2.4). There is also additional information indicating the clause in which the subgoal with the entry constraint occurs and the position of the subgoal within that clause. The procedure-entry and procedure-exit operations are

split up into smaller operations, similar to the ones indicated in Section 3.2.3.2 (e.g. procedure-entry first projects the abstract call constraint of the subgoal onto the subgoal variables, yielding the entry constraint; then the entry constraint is used to compute the in-constraint at the beginning of each clause used to resolve the subgoal). For a detailed explanation, we refer to [96, 94].

The information in the memo table serves two purposes : (1) it is used to avoid non-termination when analysing recursive predicates and (2) it is used to improve efficiency by avoiding repeated subcomputations; subcomputations involving the same goal and entry constraint require only a simple table lookup to obtain the associated exit constraint.

- When dealing with recursive predicates, the non-recursive clauses are analysed first, thus providing a better first approximation for the exit constraint of a recursive call than $\perp$ (which would be obtained if the recursive clauses were treated first). This reduces the number of fixpoint iterations, speeding up the analysis.

- Information on dependencies between predicate calls and on previous lookups in the memo table during analysis of a predicate call is used to reanalyse the necessary program parts only when some of the looked-up information has changed (i.e. when the information at some point involved in a fixpoint computation has changed).

The PLAI system computes specialised versions (also referred to as multi-variants) for each predicate : a call that is syntactically different from all previously encountered calls or that has a different entry constraint gives rise to a new predicate version. The system thus performs a quite detailed analysis. In some cases, the above specialisation criterion turns out to be too strong, in the sense that it gives rise to specialised versions which are identical in spite of the different entry patterns (i.e. the different entry patterns yield the same abstract information within all clauses of the different predicate versions). An example is given in Appendix A (*vecmat* program). Specialisation could be avoided in such cases by applying a more sophisticated specialisation criterion.

The user output of the PLAI system is an annotated program, i.e. a program with the abstract constraint written out at each program point. Currently, there are two options : either the system produces all predicate versions encountered during the analysis (corresponding to different entry-exit patterns for the predicate), or it returns only one general version for each predicate by computing the least upper of the abstract constraints at each program point (the output then consists of the original program augmented with the annotations at each program point).

As mentioned in [43, 41], the generalisation of the original PLAI system to support the analysis of *constraint* logic programs required only very small modifications to the overall algorithm. So almost all of the existing implementation was reused. Of course, for each particular analysis, the domain-dependent abstract operations have to be implemented and incorporated into the system.

10.3 Freeness analysis

In this section, we point out various implementation issues concerning the (minimal) freeness analysis and the F^* and M^* parts of resp. the $\mathcal{DF}$ and $\mathcal{DM}$ analyses.

Data-structures

In the PLAI system, program variables are represented as free Prolog variables. We represent a set of variables as an ordered list of variables. For a set of sets of variables, two representations are used :

1. In case of the $\mathcal{F}$ and $\mathcal{DF}$ analyses, a set of sets of variables can become quite large (due to the exhaustive enumeration of possible dependencies), hence it is implemented as a balanced tree (more precisely, a 23-tree [66, 50]).
2. In case of the $\mathcal{M}$ and the $\mathcal{DM}$ analyses, where the freeness part is minimised, a set of sets of variables is usually smaller and is represented by an ordered list (to avoid the overhead of working with a balanced tree). However, during abstract conjunction, the closure of the set (or at least a subset of it) must be computed, increasing its size considerably; whereas the set resulting from abstract conjunction is again minimised. So a local transformation to 23-trees is applied during abstract conjunction.

Abstract operations

The most important and also the most expensive abstract operation is abstract conjunction. For the minimal freeness analysis (and also for the M^* part of the $\mathcal{DM}$ analysis), this operation involves taking the closure under union of a set S of possible dependencies. As already pointed out in Section 6.3, only a part of the set S must actually be closed. The implemented conjunction algorithm gains some more efficiency by treating the singletons in S (representing non-free variables) separately as they play a specific role in the non-freeness propagation. So the singletons are not involved in the closure operation. In contrast, for the $\mathcal{F}$ and $\mathcal{DF}$ analyses, it is difficult to optimise abstract conjunction as the set of possible dependencies is already represented exhaustively.

Another point worth mentioning concerns procedure-exit. All our analysers only compute the new component of the abstract exit constraint (i.e. the answer constraint restricted to the call variables), as this is the only component needed at extension to obtain the success constraint of a procedure call. The old component is given a dummy value. The $(dummy, AC^n_{exit})$ constraint is the one that is stored as exit value in the entry-exit table for procedure calls. If entry-exit patterns for calls should be returned, the correct exit pattern $(AC^o_{exit}, AC^n_{exit})$ is obtained by looking up $(AC^o_{entry}, AC^n_{entry})$ and $(dummy, AC^n_{exit})$ in the table and computing AC^o_{exit} as $AC^o_{exit} = AC^t_{entry} \cup (AC^t_{entry} \oplus^{\mathcal{F}} AC^n_{exit})$ (or using the corresponding abstract conjunction operations for the $\mathcal{DF}$, $\mathcal{M}$ or $\mathcal{DM}$ analyses). The full computation of the abstract exit constraint is then done only once, namely when producing the user output and not during the analysis which may involve several fixpoint iterations.

Abstraction of a constraint

The definition of constraint abstraction as presented in Chapters 5, 6, 7 and 8 involves computing the solved form of the constraint. If the abstract interpretation system were implemented in a CLP language, the solved form algorithm would come for free. However, it is not available in the PLAI system written in Prolog and should thus be provided by the

developer of the analysis. An alternative solution requiring less programming effort is to abstract only *primitive* constraints; the abstraction of a constraint C can then be computed by joining (via abstract conjunction) the abstractions of the primitive constraints in C. This is the approach followed in our current prototype. A disadvantage of the approach is the possible loss of precision. For example, consider $C \equiv X+Y=Z \wedge X+Y=T$; abstracting C as a whole (using Definition 5.2.10) yields $\alpha^{\mathcal{F}}(C) = close(\{\{X,Y,Z\},\{X,Y,T\},\{Z,T\}\})$, whereas computing the abstraction of the primitive constraints in C and joining them via $\Lambda^{\mathcal{F}}$ gives $\alpha^{\mathcal{F}}(C) = \alpha^{\mathcal{F}}(X+Y=Z)\ \Lambda^{\mathcal{F}}\alpha^{\mathcal{F}}(X+Y=T) = \{\{X,Y,Z\}\}\ \Lambda^{\mathcal{F}}\{\{X,Y,T\}\}$ $= close(\{\{X,Y,Z\},\{X,Y,T\},\{X,Z,T\},\{Y,Z,T\},\{Z,T\}\})$; note that the latter is less precise. An advantage is that abstract information is also available in between primitive constraints. This more detailed information is needed for low-level compiler optimisations. The ideal approach is a combination of the two : the abstraction of primitive constraints can be used to supply information in between primitive constraints; afterwards, the more precise abstraction of the entire constraint (i.e. the conjunction of the primitive constraints) can be computed using the solved form algorithm, thus providing more precise call information to the literal following the constraint.

Normalisation

Two sets of analysers are currently available : analysers for normalised programs and analysers for non-normalised programs. The former expect input programs that are already in normalised form. So the program normalisation is a pre-processing step which is not included in the actual analysis. The latter can handle both normalised and non-normalised programs.

In the analysers for normalised programs, normalisation is even further enforced than the call and head normalisation and the splitting of composite constraints (constraints including a mixture of Herbrand, numerical or tuple functors) Primitive constraints are restricted to be of one of the following forms :

- $X = t$ with X variable and t a Herbrand term;
- $a_1X_1 + \ldots + a_nX_n \Diamond b$ resp. $a_1t_1 + \ldots + a_nt_n \Diamond b$ for linear resp. non-linear numerical constraints (the X_i are distinct variables, the a_i are non-zero numbers, b is a number, $\Diamond \in \{=, \neq, >, \geq\}$ and the t_i are of the form $Y_1 \odot \ldots \odot Y_m$ with $\odot \in \{*, /\}$ and the Y_i being (not necessarily distinct) variables);
- $X = t$ with X a variable and t a PrologIII tuple term;
- $X :: N$ with X and N distinct variables.

In contrast, the analysers for non-normalised programs require less normalisation than mentioned in Section 9.1, since abstraction is performed on *primitive* constraints instead of full constraints. More precisely, the *local* normalisation during constraint abstraction is avoided. For example, the primitive constraint $X = f(Y+Z, T)$ is directly abstracted as $\alpha^{\mathcal{F}}(X = f(Y+Z, T)) = close(\{\{X\},\{X,Y,Z\},\{X,T\}\})$, without normalisation.

Features actually implemented

The features that are actually implemented in the prototype are the basic features of Chapters 5, 6, 7 and 8 and the following extensions :

- treatment of non-normalised programs;
- better treatment of non-linear constraints (Definition 9.3.1, Algorithm 9.3.2), at least for the $\mathcal{M}$ and $\mathcal{DM}$ analyses (not for the $\mathcal{F}$ and $\mathcal{DF}$ ones);

- space-optimisation of abstract constraints, at least for the $\mathcal{M}$ and $\mathcal{DM}$ analyses (cf. Sections 6.4.5.2 and 8.4.5.2);
- inference of definite/possible failure;
- abstraction of PrologIII tuples.

10.4 Combined definiteness-freeness analysis

The $\mathcal{DF}$ and $\mathcal{DM}$ analyses use definiteness information that is provided by the definiteness analysis of [43, 41]. This analysis is also implemented within the PLAI system (the implementation was carried out simultaneously with the implementation of the freeness analysis in the context of the ESPRIT project PRINCE). The developers of the PLAI system also extended the system to enable the combination of the definiteness and freeness analysis. The actual way of communication between both analysers was studied in collaboration between U.P.Madrid and K.U.Leuven; the implementation of the $\mathcal{DF}$ and $\mathcal{DM}$ abstract operations was performed at K.U.Leuven.
We now describe the basic aspects of the combination. The set of definite variables (approximating the set of actual definite variables) is a separate component in the abstract constraint of the definiteness analysis. The definiteness and freeness analysis (i.e. $\mathcal{DF}$ or $\mathcal{DM}$ analysis) are executed in parallel. At each point of the analysis (i.e. at the application of one of the higher-level abstract operations), the definiteness operation is called first. Afterwards, the set of definite variables is extracted from the result of that operation and passed as an extra parameter to the freeness operation. If the definiteness operation results in the abstract constraint $\bot$, the freeness operation proceeds with $\bot$. So, information is always passed from the definiteness to the freeness analysis; information passing in the other direction is restricted to the passing of $\bot$ information : if a freeness operation yields $\bot$ where the preceding definiteness operation did not give $\bot$, the subsequent definiteness analysis continues with $\bot$ (thus computation of useless information is avoided). As a result of executing both analysers in parallel, the freeness analysis is required to be carried out along with the definiteness analysis and vice versa; so if one of the analyses requires more fixpoint iterations than the other, this may have a negative effect on the efficiency of the combined execution. In other words, the execution time of the combined analysis (definiteness and $\mathcal{DF}$ or $\mathcal{DM}$ in parallel) may be larger than the sum of the execution times of the separate analyses (definiteness and $\mathcal{F}$ or $\mathcal{M}$), as will be shown in Section 11.2. The extra iterations of the $\mathcal{DF}$ and $\mathcal{DM}$ analyses due to additional iterations of the definiteness analysis could be avoided by first performing the definiteness analysis on itself, and then use the programs annotated by the definiteness analysis as input for the freeness analysis.

Chapter 11

Results

In this chapter we evaluate the efficiency and accuracy of the different mode analyses. We also indicate how the derived information can be used to optimise CLP programs.

11.1 Benchmarks

A relatively wide range of benchmarks has been used in our experiments. We describe their purpose and their entry pattern(s). For each benchmark there are two program versions : a normalised one, in which all constraints are made explicit and are in a particular form (cf. Section 10.3), and a non-normalised one. As already mentioned in Section 9.1, the non-normalised version in general allows to express an entry pattern more directly and more precisely than the normalised version : it can express that an argument is a partially instantiated structure of a particular form.

- **dnf :**
  a program that converts a propositional formula into disjunctive normal form (taken from the CLP($\mathcal{R}$) distribution); the considered entry pattern is $dnf(\mathrm{d},\mathrm{f})$ where the first argument is a propositional formula and the second argument is the disjunctive normal form of that formula.

- **fib :**
  a program defining the relation $fib(N, F)$ where F is the N^{th} Fibonacci number (taken from the CLP($\mathcal{R}$) distribution); the considered entry pattern is $fib(\mathrm{d},\mathrm{f})$.

- **laplace :**
  a program to solve the Dirichlet problem for Laplace's equation in two dimensions using Leibman's five-point finite-difference approximation (program taken from the CLP($\mathcal{R}$) distribution, also described in [54]). The program computes the temperature of a surface at discrete points; it specifies that the temperature at each non-boundary point is the average of those of the four neighbouring points. A call is of the form $laplace(M)$ where M is a matrix (list of lists) of temperature values at discrete points of the surface. Two possible entry patterns are considered. The first one is $laplace1(\mathrm{d})$. The second one is $laplace2(M)$ where the argument M is a matrix of free variables which just specifies the size of the surface (i.e. a partially instantiated

argument). For an analyser that can deal only with normalised programs, this entry pattern cannot be expressed precisely; instead we have :

$$main :- V = matrix\_of\_free\_variables,\ laplace(V).$$

This in fact leads to the less precise entry pattern $laplace(\mathbf{a})$. Whereas if the analyser is able to deal with non-normalised programs, the entry pattern can be represented directly and precisely via $laplace(matrix\_of\_\mathbf{f}\_variables)$.

- **listlength :**
  a program that specifies the relation between a list and its length; the considered entry pattern is $listlength(\mathbf{d}, \mathbf{f})$, where the first argument is a list and the second argument is its length.

- **meal :**
  a program to compute a balanced meal (program taken from the PrologIII distribution; also described in [19]); the considered entry pattern is $lightMeal(\mathbf{f}, \mathbf{f}, \mathbf{f})$ where the first argument represents the first dish, the second argument is the main course and the last argument is the dessert.

- **mining :**
  a PrologIII program that optimises the revenue of an open mine (program written by Eric Vetillard, PrologIA – original problem from H. P. Williams); the considered entry pattern is $mining(\mathbf{f}, \mathbf{f})$ where the first argument is the mine topology and the second argument is the revenue of the mine.

- **mortgage :**
  the famous mortgage program (taken from the CLP($\mathcal{R}$) distribution; also described in [83]). A call is of the form $mortgage(P, T, I, B, MP)$ where P is the principal, T is the life of the mortgage (in months), I is the fixed (but compounded) monthly interest rate, B is the outstanding balance at the end and MP is the monthly payment. Several entry patterns are considered : $mortgage1(\mathbf{a}, \mathbf{a}, \mathbf{a}, \mathbf{a}, \mathbf{f})$, $mortgage2(\mathbf{a}, \mathbf{a}, \mathbf{a}, \mathbf{f}, \mathbf{a})$ and $morigage3(\mathbf{f}, \mathbf{d}, \mathbf{d}, \mathbf{d}, \mathbf{d})$.

- **num :**
  a PrologIII program to transform numbers into a sequence of letters and phonemes (program written by Eric Vetillard, PrologIA); the considered entry pattern is $nombre(\mathbf{d}, \mathbf{f}, \mathbf{f})$ where the first argument is a number, the second argument is the corresponding sequence of letters and the last argument is the corresponding sequence of phonemes. This program uses specific features of PrologIII tuples which cannot be obtained using ordinary Prolog lists.

- **power :**
  a PrologIII program that minimises the production cost of power stations (program written by Eric Vetillard, PrologIA); the considered entry pattern is $pow(\mathbf{f})$ where the argument is the production cost of the power stations.

- **rectangle :**
  a PrologIII program to fill a rectangle of dimension $1 \times b$ with n squares (program

taken from the PrologIII distribution; also described in [19]); the considered entry pattern is *fillRectangle*(f, a) where the first argument denotes the width b of the rectangle and the second argument is a PrologIII tuple of length n (when the program finishes, the first argument b is instantiated to a number (the width of the rectangle) and the tuple argument contains the sizes of the n squares that fill the rectangle).

- **runkut :**
  a program for first-order ordinary differential equation solving, using the runge-kutta method (program written by W. Krautter, FAW Ulm); the considered entry pattern is *solve*(d, d, f) where the first argument is the initial pair of x-y values, the second argument is the step size and the last argument is the final x-value.

- **sendmm :**
  the well-known *send* + *more* = *money* puzzle (program taken from the PrologIII distribution); the considered entry pattern is *solution*(f, f, f) where the arguments are the numbers assigned to resp. *send*, *more* and *money*.

- **sumlist :**
  a program that specifies the relation between a list of numbers and the sum of its elements; the considered entry pattern is *sumlist*(d, f) where the first argument is a list and the second argument is the sum of its elements.

- **trap :**
  a program for first-order ordinary differential equation solving, using the trapezoidal method (program written by W. Krautter, FAW Ulm); the considered entry pattern is *solve*([d, d], d, [d, f]) where the first argument is the initial pair of x-y values, the second argument is the number of steps and the third argument is the final pair of x-y values. For the normalised analysis, the entry pattern cannot be expressed directly nor precisely and is given via

  main($T1, N, Xn$) : $-$ $T2 = [Xn, Yn]$, *solve*($T1, N, T2$).
  with entry pattern *main*(d, d, d).

  This leads to a less precise entry pattern for solve : *solve*(d, d, a).

- **vecmat :**
  a PrologIII program (written by W. Krautter, FAW Ulm) that performs vector and matrix operations, more precisely vector addition (*vecadd*), multiplication of a matrix and a vector (*matvecmul*) and matrix multiplication (*matmul*). A vector is represented as a PrologIII tuple, a matrix is a tuple of tuples. Two series of entry patterns are considered : *vecmat*1 which gives rise to *matvecmul*(d, d, f), *vecadd*(f, d, d) and *matmul*(d, d, f); and *vecmat*2 which gives rise to *matvecmul*(f, d, d), *vecadd*(f, d, a) and *matmul*(d, f, d).

The size of the programs is indicated by means of the number of user-defined predicates (Pred), the number of clauses (Cl) and the number of calls (Calls) to user-defined predicates (this includes also the entry call); these numbers are given in Table 11.1. For *trap* and *laplace*2, the "+1" applies to the normalised version of the benchmark in which an extra predicate *main* is needed to express the call pattern.

Program	Pred	Cl	Calls
dnf	3	32	40
fib	1	2	3
laplace1	2	4	4
laplace2	2(+1)	4(+1)	4(+1)
listlength	1	2	2
meal	6	11	6
mining	25	50	41
mortgage1	1	2	2
mortgage2	1	2	2
mortgage3	1	2	2
num	17	97	54
power	18	42	29
rectangle	5	10	10
runkut	4	5	8
sendmm	4	7	7
sumlist	1	2	2
trap	4(+1)	5(+1)	6(+1)
vecmat1	8	15	15
vecmat2	8	15	15

Table 11.1: Size of the benchmarks

Program	Rec	TailRec	NonRec
dnf	2	1	0
fib	1	0	0
laplace1	0	2	0
laplace2	0	2	0(+1)
listlength	0	1	0
meal	0	0	6
mining	3	11	11
mortgage1	0	1	0
mortgage2	0	1	0
mortgage3	0	1	0
num	0	0	17
power	0	9	9
rectangle	2	2	1
runkut	0	1	3
sendmm	0	3	1
sumlist	0	1	0
trap	0	1	3(+1)
vecmat1	0	7	1
vecmat2	0	7	1

Table 11.2: Recursivity in the benchmarks

Program (norm)	Var	MaxVar	AvgVar	ProgPt	DifCalls
dnf	133	11	4.2	130	3
fib	10	6	3.3	13	1
laplace1	36	18	9	20	2
laplace2	53	18	10.6	23	3
listlength	7	5	3.5	7	1
meal	26	6	2.4	33	6
mining	238	24	4.8	260	25
mortgage1	12	7	6	8	1
mortgage2	12	7	6	8	1
mortgage3	12	7	6	8	1
num	382	10	3.9	429	17
power	205	20	4.9	191	18
rectangle	55	13	5.5	49	5
runkut	45	15	9	30	4
sendmm	26	12	3.7	30	4
sumlist	7	5	3.5	7	1
trap	55	11	9.2	35	5
vecmat1	76	11	5.1	63	8
vecmat2	81	11	5.4	65	8

Program (nonnorm)	Var	MaxVar	AvgVar	ProgPt	DifCalls
dnf	75	7	2.3	72	14
fib	3	3	1.5	6	3
laplace1	24	12	6	8	4
laplace2	43	16	8.6	11	4
listlength	4	4	2	4	1
meal	10	6	0.9	20	6
mining	126	18	2.5	151	32
mortgage1	8	5	4	4	2
mortgage2	8	5	4	4	2
mortgage3	8	5	4	4	2
num	233	10	2.4	280	17
power	137	19	3.3	122	24
rectangle	33	9	3.3	27	8
runkut	31	9	6.2	16	6
sendmm	19	11	2.7	23	6
sumlist	4	4	2	4	1
trap	32	9	6.4	14	5
vecmat1	44	8	2.9	31	9
vecmat2	47	8	3.1	31	10

Table 11.3: Additional properties of the benchmarks

Some other measures that provide an insight into the complexity of each benchmark and that are important in the evaluation of the analysers are given in Tables 11.2 and 11.3. Table 11.2 gives the number of recursive predicates (Rec) that are not tail-recursive, the number of tail-recursive predicates (TailRec) and the number of non-recursive predicates (NonRec). Recursivity is a property that has a large influence on the time and space consumption of the analysis : the abstract interpretation algorithm (Algorithm 3.2.1) performs a fixpoint computation for each recursive predicate, computing successive approximations of its exit pattern. Hence programs containing recursive predicates lead to a more complex analysis than non-recursive programs. It is also important to know whether a predicate is tail-recursive or not. For a tail-recursive call, there are no subsequent program points in the clause that are influenced by the call's exit pattern. Whereas in the non-tail-recursive case, the successive approximations of the exit pattern of the predicate may give rise to successive approximations of the abstract constraints at the program points following a non-tail-recursive call. This may cause the creation and analysis of useless predicate versions for the calls following a non-tail-recursive call.

Other relevant program properties are the number of variables in the program (Var), the maximum and average number of variables in the program clauses (MaxVar and AvgVar), the number of program points (ProgPt) and the number of syntactically different calls (DifCalls). These figures differ for the normalised and non-normalised versions of the benchmarks and are given in Table 11.3. Due to the normalisation, the Var, MaxVar, AvgVar and ProgPt figures are larger for the normalised benchmarks compared to the non-normalised ones. However, the number DifCalls is usually smaller. In fact, the number DifCalls for the normalised benchmarks is identical to the number of predicates, as the normalisation implies that all calls have distinct variables as arguments.
The number of variables in a clause is a measurement for the size of the abstract constraints adorning the program points in the clause, which in turn influences the efficiency of the abstract operations. The number of syntactically different calls is important with regard to the criterion for specialisation used in the abstract interpretation algorithm : a new subanalysis is started for a call that is syntactically different from previously encountered calls to the same predicate. Whereas if a call is syntactically identical to a previous call and has the same abstract entry constraint, a simple lookup of memoised information is performed. For recursive predicates, a larger number of syntactically different predicate calls implies a larger number of fixpoint computations or a longer chain of calls in one computation.

11.2 Efficiency results

In this section we study the performance of the different mode analysers, both with respect to time and memory consumption. The efficiency results depend on (1) the size of the AND-OR-graph (in which the number of predicate versions is an important factor) and the number of fixpoint iterations (cf. Table 11.6) and (2) the size of the abstract constraints and the kind and size of variable dependencies therein. The numbers in Tables 11.2 and 11.3 that are related to the first issue are DifCalls, Rec and TailRec. For the second issue, the numbers MaxVar and AvgVar are important. Also, two classes of inputs (i.e. programs and associated entry patterns) can be distinguished :

1. inputs that from the start give rise to a lot of definite variables and related dependencies (dnf, fib, laplace1, listlength, meal, sumlist, vecmat1). They create small sets of possible dependencies; also the dependencies themselves are small (at least in case of the $\mathcal{DF}$ and $\mathcal{DM}$ analyses which exploit definiteness information).

2. inputs that do not allow to infer much definite information or infer it only towards the end of the program. They establish large sets of possible dependencies and often also large dependencies (laplace2, mining, mortgage1, mortgage2, mortgage3, num, power, rectangle, runkut, sendmm, trap, vecmat2).

Table 11.4 shows the timings obtained for the different analysers (SUN Sparc 2, SICStus 2.1 - fastcode, average out of 10 runs, including the time for garbage collection and stack shifts); "–" indicates that the analyser did not produce a result (ran out of memory). The indication "norm" stands for normalised, "nonnorm" for non-normalised. Associated to an analyser, "norm" means that the analyser can only deal with normalised programs; "nonnorm" refers to a general analyser which can handle both normalised and non-normalised programs (note : the $\mathcal{F}$ and $\mathcal{DF}$ abstractions have only been implemented for normalised programs).
Note that the programs for which the analysers do not give a result are the ones with the largest maximum and average number of variables in the program clauses (cf. Table 11.3). This implies that the abstract constraints in these clauses may become quite large, thus giving problems at abstract conjunction.

The speed of the analysers is compared in Table 11.5. Each figure gives the percentage of time taken by an optimised analysis with respect to the weaker analysis it is compared to; "Inf" indicates that the optimised analysis is definitely better than the weaker one, since the latter did not give a result. A "–" indicates that no comparison can be made, because even the optimised analysis did not produce a result.

The upper part of Table 11.5 (containing the figures for the norm-analysers on the norm-programs) shows the performance improvements obtained by the optimisations of the original freeness abstraction, being the exploitation of definiteness information, the minimisation and the combination of both. Columns 1 and 2 consider the improvement of the $\mathcal{DF}$ abstraction (i.e. splitting off the definite part from the original freeness abstraction), first with respect to the freeness part only and secondly with respect to the total mode analysis. Column 3 indicates the effect of minimisation on the freeness analysis. Columns 4 and 5 resp. show the extra improvement of splitting off the definite part with respect to the minimised freeness part, and of minimising the freeness part after splitting off the definite part. Columns 6, 7 and 8 indicate the total improvement obtained by the $\mathcal{DM}$ abstraction (which combines the optimisations) over the original freeness part only and over the total mode analyses $\mathcal{D}+\mathcal{M}$ and $\mathcal{D}+\mathcal{F}$.
The figures show that the optimisations yield a substantial speed-up and result in a practical mode analysis system. For the first class of inputs that from the start give rise to a lot of definite variables and related dependencies (cf. above the double line in the norm-part of Table 11.5), the approach that pays off best is splitting off the definite part and thereby almost completely reducing the freeness part. The second class of inputs (cf. below the double line in the norm-part of Table 11.5) establish larger sets of possible dependencies. In that case, the minimising approach ($\mathcal{M}$) is the most effective one. For this class of

Program (norm)	Analysis times (seconds)				
	$\mathcal{D}$	$\mathcal{F}$	$\mathcal{DF}$	$\mathcal{M}$	$\mathcal{DM}$
dnf	0.466	119.467	1.188	2.298	1.136
fib	0.045	0.140	0.060	0.060	0.056
laplace1	0.070	–	0.118	216.099	0.108
listlength	0.032	0.053	0.040	0.036	0.032
meal	0.114	0.191	0.140	0.140	0.140
sumlist	0.028	0.066	0.037	0.048	0.030
vecmat1	0.262	388.191	0.426	0.600	0.416
laplace2	9.298	–	–	–	–
mining	2.397	–	–	82.765	145.093
mortgage1	0.066	13.198	4.461	0.076	0.140
mortgage2	0.064	2.973	2.096	0.050	0.104
mortgage3	0.040	13.101	0.094	0.074	0.077
num	10.957	44.589	12.369	10.995	11.633
power	2.439	–	–	19.157	9.024
rectangle	10.718	330.771	569.227	1.601	12.275
runkut	0.148	–	0.957	0.449	0.226
sendmm	31.449	68.247	95.653	11.531	45.934
trap	6.258	–	–	0.538	7.512
vecmat2	0.542	530.053	28.675	0.874	1.519

Program (nonnorm)	Analysis times (seconds)		
	$\mathcal{D}$	$\mathcal{M}$	$\mathcal{DM}$
dnf	1.056	7.036	3.715
fib	0.038	0.096	0.082
laplace1	0.058	–	0.124
listlength	0.010	0.024	0.022
meal	0.036	0.084	0.077
sumlist	0.009	0.038	0.019
vecmat1	0.114	0.790	0.329
laplace2	1.841	24.615	26.019
mining	2.741	12.439	15.113
mortgage1	0.171	0.158	0.439
mortgage2	0.172	0.110	0.329
mortgage3	0.026	0.168	0.118
num	1.212	1.988	2.085
power	5.775	19.015	9.180
rectangle	11.213	3.797	15.241
runkut	0.058	0.509	0.157
sendmm	11.949	6.964	20.813
trap	3.443	0.872	4.595
vecmat2	0.439	0.887	1.231

Table 11.4: Timings of the norm- and nonnorm-analysers

Program (norm)	$\frac{DF-D}{F}$	$\frac{DF}{F+D}$	$\frac{M}{F}$	$\frac{DM-D}{M}$	$\frac{DM}{DF}$	$\frac{DM-D}{F}$	$\frac{DM}{D+M}$	$\frac{DM}{D+F}$
dnf	0.0060	0.0099	0.0192	0.2916	0.9562	0.0056	0.4110	0.0095
fib	0.1071	0.3243	0.4286	0.1833	0.9333	0.0786	0.5333	0.3027
laplace1	Inf	Inf	Inf	0.0002	0.9153	Inf	0.0005	Inf
listlength	0.1509	0.4706	0.6792	0.0000	0.8000	0.0000	0.4706	0.3765
meal	0.1361	0.4590	0.7330	0.1857	1.0000	0.1361	0.5512	0.4590
sumlist	0.1364	0.3936	0.7273	0.0417	0.8108	0.0303	0.3947	0.3191
vecmat1	0.0004	0.0011	0.0015	0.2567	0.9765	0.0004	0.4826	0.0011
laplace2	–	–	–	–	–	–	–	–
mining	–	–	Inf	1.7241	Inf	Inf	1.7037	Inf
mortgage1	0.3330	0.3363	0.0058	0.9737	0.0314	0.0056	0.9859	0.0106
mortgage2	0.6835	0.6902	0.0168	0.8000	0.0496	0.0135	0.9123	0.0342
mortgage3	0.0041	0.0072	0.0056	0.5000	0.8191	0.0028	0.6754	0.0059
num	0.0317	0.2227	0.2466	0.0615	0.9405	0.0152	0.5299	0.2094
power	–	–	Inf	0.3437	Inf	Inf	0.4179	Inf
rectangle	1.6885	1.6669	0.0048	0.9725	0.0216	0.0047	0.9964	0.0359
runkut	Inf	Inf	Inf	0.1737	0.2362	Inf	0.3786	Inf
sendmm	0.9408	0.9594	0.1690	1.2562	0.4802	0.2122	1.0687	0.4607
trap	–	–	Inf	2.3309	Inf	Inf	1.1054	Inf
vecmat2	0.0531	0.0540	0.0016	1.1178	0.0530	0.0018	1.0727	0.0029

Program (nonnorm)	$\frac{DM-D}{M}$	$\frac{DM}{D+M}$
dnf	0.3779	0.4591
fib	0.4583	0.6119
laplace1	Inf	Inf
listlength	0.5000	0.6471
meal	0.4881	0.6417
sumlist	0.2632	0.4043
vecmat1	0.2722	0.3639
laplace2	0.9823	0.9835
mining	0.9946	0.9956
mortgage1	1.6962	1.3343
mortgage2	1.4273	1.1667
mortgage3	0.5476	0.6082
num	0.4391	0.6516
power	0.1791	0.3703
rectangle	1.0608	1.0154
runkut	0.1945	0.2769
sendmm	1.2728	1.1005
trap	1.3211	1.0649
vecmat2	0.8929	0.9284

Table 11.5: Speed comparison of the norm- and nonnorm-analysers

Program (norm)	$\mathcal{D}$		$\mathcal{F}$		$\mathcal{DF}$		$\mathcal{M}$		$\mathcal{DM}$	
	EE	FIX	EE	FIX	EE	FIX	EE	FIX	EE	FIX
dnf	3	3	5	5	5	5	5	5	5	5
fib	1	1	1	1	1	1	1	1	1	1
laplace1	2	2	–	–	2	2	2	2	2	2
listlength	1	1	1	1	1	1	1	1	1	1
meal	6	0	6	0	6	0	6	0	6	0
sumlist	1	1	1	1	1	1	1	1	1	1
vecmat1	8	7	10	9	10	9	10	12	10	9
laplace2	3	2	–	–	–	–	–	–	–	–
mining	31	33	–	–	–	–	26	20	31	33
mortgage1	1	2	2	2	2	3	2	2	2	3
mortgage2	1	2	1	1	1	2	1	1	1	2
mortgage3	1	1	2	2	2	2	2	2	2	2
num	22	0	17	0	22	0	17	0	22	0
power	24	22	–	–	–	–	18	12	24	22
rectangle	7	12	6	5	7	13	6	9	7	13
runkut	4	1	–	–	4	1	4	2	4	1
sendmm	4	5	4	3	4	5	4	4	4	5
trap	9	3	–	–	–	–	5	2	9	3
vecmat2	10	14	12	11	15	21	12	14	15	21

Program (nonnorm)	$\mathcal{D}$		$\mathcal{M}$		$\mathcal{DM}$	
	EE	FIX	EE	FIX	EE	FIX
dnf	14	14	26	26	26	26
fib	3	3	3	3	3	3
laplace1	4	4	4	7	4	4
listlength	1	1	1	1	1	1
meal	6	0	6	0	6	0
sumlist	1	1	1	1	1	1
vecmat1	9	8	11	13	11	10
laplace2	4	4	4	4	4	4
mining	36	39	32	28	36	39
mortgage1	2	2	3	3	3	3
mortgage2	2	2	2	2	2	2
mortgage3	2	2	3	3	3	3
num	22	0	17	0	22	0
power	28	27	24	18	28	27
rectangle	11	21	9	14	11	22
runkut	6	1	6	2	6	2
sendmm	6	7	6	6	6	7
trap	11	5	5	3	11	5
vecmat2	12	16	13	15	16	22

Table 11.6: FIX and EE of the norm- and nonnorm-analysers

benchmarks, the combination of the freeness and definiteness analyses sometimes causes overhead, in the sense that the gain by exploiting definiteness information does not outweigh the increase in the number of predicate versions and/or number of fixpoint iterations (cf. the figures that are larger than 1 in Table 11.5). In those cases, either the $\mathcal{DF}$ time is worse than the sum of the $\mathcal{D}$ and $\mathcal{F}$ times (*rectangle*), or the $\mathcal{DM}$ result is worse than the sum of the $\mathcal{D}$ and $\mathcal{M}$ times (*mining*, *sendmm*, *trap*, *vecmat*2). In order to have a better understanding of these results, the number of entry-exit patterns (EE) and the number of fixpoint iterations (FIX) of the program analyses are given in the upper part of Table 11.6. The number FIX is the sum of all passes through the recursive predicates in the program. In most of the cases that stand out (*rectangle*, *mining*, *sendmm*, *trap*), the definiteness analysis performs more iterations before reaching its fixpoint than the freeness analysis; mostly also the number of entry-exit patterns is larger. When combining the two analyses, the $\mathcal{DF}$ or $\mathcal{DM}$ analysis has to perform at least as many iterations as the definiteness analysis, however now not only computing the definite part but also taking into account the (reduced) freeness part. If the latter is still quite large after splitting off the definite information, the overhead of the extra iterations outweighs the effect of reducing the freeness part. For the *vecmat*2 benchmark, the number of entry-exit patterns and fixpoint iterations in the $\mathcal{DM}$ analysis is larger than in the $\mathcal{D}$ or $\mathcal{M}$ analysis. Again, the extra iterations outweigh the effect of reducing the freeness part.
For any input, combining the optimisations results in an efficient analysis compared to separately running the definiteness and *original* freeness analyser (cf. last column in the norm-part of Table 11.5) : the speed-up is at least 54% (except for *laplace*2).

The lower parts of Tables 11.5 and 11.6 give similar results for the analysis of *non-normalised* programs. Table 11.5 indicates the extra improvement of splitting off the definite part with respect to the minimised freeness part, and the total improvement obtained by the $\mathcal{DM}$ abstraction compared to the sum of separate $\mathcal{D}$ and $\mathcal{M}$ analysis times. For *rectangle*, *sendmm* and *trap*, the $\mathcal{DM}$ time is larger than the $\mathcal{D}+\mathcal{M}$ time due to the increase in the number of predicate versions and/or fixpoint iterations of $\mathcal{DM}$ w.r.t. $\mathcal{M}$. This increase outweighs the gain obtained by exploiting definiteness information. For *mortgage*1 and *mortgage*2, the number of versions and iterations is the same, but still the $\mathcal{DM}$ time is larger than the $\mathcal{D}+\mathcal{M}$ time. Recall that the timings include the time for garbage collection and stack expansions. The $\mathcal{D}$ and $\mathcal{M}$ analysis involve no stack expansions, whereas the $\mathcal{DM}$ analysis requires an expansion of the global stack (and control stack in case of *mortgage*1). The time for this operation makes out the difference between the $\mathcal{DM}$ and $\mathcal{D}+\mathcal{M}$ times.

The memory consumption of the analyses is given in Table 11.7. It gives the maximal space that was needed during analysis. Again, the optimised analyses ($\mathcal{DF}$, $\mathcal{M}$ and $\mathcal{DM}$) are much better than the original freeness analysis; the $\mathcal{DM}$ analysis which combines the use of definiteness information and the minimisation in most cases gives the best results (compared with exploiting just one optimisation, i.e. compared with $\mathcal{DF}$ or $\mathcal{D}+\mathcal{M}$). In general, the sum of the memory consumed by the $\mathcal{D}$ and $\mathcal{F}$ analyses is larger than the memory used by the $\mathcal{DF}$ analysis; a similar result holds for the comparison of $\mathcal{DM}$ to the sums of $\mathcal{D}$ and $\mathcal{M}$. There is only one figure that stands out, namely the $\mathcal{DM}$ one for the *mining* benchmark in the norm-part of Table 11.7. This is due to the fact that for $\mathcal{DM}$ the freeness analysis is forced to iterate along with the definiteness analysis; this outweighs the

Program (norm)	Memory consumption (MByte)				
	$\mathcal{D}$	$\mathcal{F}$	$\mathcal{DF}$	$\mathcal{M}$	$\mathcal{DM}$
dnf	3.124786	21.557770	4.124542	4.124542	4.124542
fib	2.562408	2.562408	2.562408	2.562408	2.562408
laplace1	2.562408	–	2.562408	66.109406	2.562408
listlength	2.562408	2.562408	2.562408	2.562408	2.562408
meal	2.562408	2.562408	2.562408	2.562408	2.562408
sumlist	2.562408	2.562408	2.562408	2.562408	2.562408
vecmat1	2.562408	66.109406	3.124786	3.124786	3.124786
laplace2	4.062042	–	–	–	–
mining	4.062042	–	–	18.058624	34.054718
mortgage1	2.562408	6.623932	4.124542	2.562408	2.562408
mortgage2	2.562408	4.124542	4.124542	2.562408	2.562408
mortgage3	2.562408	6.623932	2.562408	2.562408	2.562408
num	4.062042	7.061310	4.062042	4.062042	4.062042
power	4.124542	–	–	5.124054	4.124542
rectangle	4.062042	73.545074	77.544098	4.062042	4.062042
runkut	2.562408	–	4.062042	3.062286	2.562408
sendmm	4.062042	18.121124	18.058624	6.061554	6.061554
trap	4.062042	–	–	3.062286	4.062042
vecmat2	3.124786	73.107697	11.622711	3.124786	4.124542

Program (nonnorm)	Memory consumption (MByte)		
	$\mathcal{D}$	$\mathcal{M}$	$\mathcal{DM}$
dnf	4.374542	4.374542	4.374542
fib	2.624969	2.624969	2.624969
laplace1	2.624969	–	2.624969
listlength	2.624969	2.624969	2.624969
meal	2.624969	2.624969	2.624969
sumlist	2.624969	2.624969	2.624969
vecmat1	2.874908	3.374786	2.874908
laplace2	4.374542	10.373077	10.373077
mining	4.374542	6.374054	6.374054
mortgage1	2.624969	2.624969	2.874908
mortgage2	2.624969	2.624969	2.874908
mortgage3	2.624969	2.624969	2.624969
num	4.374542	4.374542	4.374542
power	4.374542	6.374054	4.374542
rectangle	4.374542	4.374542	4.374542
runkut	2.624969	2.874908	2.874908
sendmm	4.374542	4.374542	6.374054
trap	4.374542	3.374786	4.374542
vecmat2	3.374786	3.374786	4.374542

Table 11.7: Memory consumption of the norm- and nonnorm-analysers

gain obtained by exploiting definiteness information. Note that the differences in memory consumption are not as large as for the time consumption. The reason is that expansion of the stacks in the SICStus system involves a fixed chunk of memory.

The impact of normalisation can be studied by comparing the $\mathcal{D}$, $\mathcal{M}$ and $\mathcal{DM}$ columns in the norm-part of Table 11.4 (obtained by applying the normalised analysis to the normalised benchmarks) with the corresponding columns in the nonnorm-part of Table 11.4 (obtained by applying the non-normalised analysis onto the non-normalised benchmarks). Since non-normalised programs in general involve less variables in a program clause, the size of the abstract constraints is usually smaller and hence the abstract operations (especially abstract conjunction) should benefit from that. However, the number of different calls is mostly larger compared with normalised programs. Especially for recursive predicates, the number of different entry patterns may have a large influence on the efficiency. A larger number of entry patterns implies a larger number of fixpoint computations or a longer chain of calls within a single fixpoint computation (note that the number of entry patterns could be decreased by using a better specialisation criterion in the abstract interpretation procedure). As a consequence of all this, the impact of normalisation is not easily predictable. The figures in Table 11.4 show that for 57.9% of the benchmarks the non-normalised analysis is better than the normalised one. Concerning memory consumption (cf. Table 11.7), the difference between the two types of analyses is less pronounced. For 63.2% of the benchmarks the normalised and non-normalised analyses consume roughly the same amount of memory (the small difference is due to the fact that for the non-normalised analysers the program space is larger, since these analysers include additional features such as an X-interface, and the code size of the abstract operations is also larger). The normalised analysis is better for 21% of the benchmarks, whereas the non-normalised one is better for 15.8 % of the programs.
The nonnorm-analysers are general analysers that can be applied to both non-normalised and normalised programs. However, as pointed out in Section 9.1, the application to normalised programs may incur an overhead compared to using the analysers tuned towards normalised programs. The reason is that procedure-entry and procedure-exit in the former analysis involve abstract conjunction and abstract projection, which are more costly than the renaming operation in the normalised analysis. Table 11.8 shows the timings of the norm- and nonnorm-analyser applied to the normalised benchmarks. For the $\mathcal{M}$ analysis, applying the nonnorm-analysis on normalised programs incurs an overhead in 72.2 % of the benchmarks. For the $\mathcal{D}$ analysis however, the nonnorm-analysis turns out to be faster than the norm-analysis in 57.9% of the benchmarks, which is against the expectations. One should ask the developers of the $\mathcal{D}$ analysis [43, 41] for an explanation of these figures. This effect is also propagated to the $\mathcal{DM}$ analysis.

The analysers considered so far abstract non-linear constraints in an imprecise way (at least as far as the freeness part is concerned) : all variables involved in a non-linear constraint are considered to be possibly non-free. In Section 9.3, a more precise treatment of non-linear constraints was proposed. Improving the precision has a negative effect on the efficiency, as the analysis becomes more complex (the constraint abstraction is more complex; also, the resulting dependencies are larger, which influences the efficiency of the abstract operations). This is reflected in Table 11.9, which gives the timings obtained using the imprecise abstraction and the improved one (only nonnorm-programs that involve

Program (norm)	norm-analyser			nonnorm-analyser		
	$\mathcal{D}$	$\mathcal{M}$	$\mathcal{DM}$	$\mathcal{D}$	$\mathcal{M}$	$\mathcal{DM}$
dnf	0.466	2.298	1.136	0.252	2.567	1.083
fib	0.045	0.060	0.056	0.021	0.054	0.042
laplace1	0.070	216.099	0.108	0.038	204.273	0.064
laplace2	9.298	–	–	19.557	–	–
listlength	0.032	0.036	0.032	0.015	0.038	0.026
meal	0.114	0.140	0.140	0.062	0.142	0.124
mining	2.397	82.765	145.093	4.535	75.729	135.737
mortgage1	0.066	0.076	0.140	0.166	0.150	0.549
mortgage2	0.064	0.050	0.104	0.180	0.080	0.337
mortgage3	0.040	0.074	0.077	0.020	0.146	0.102
num	10.957	10.995	11.633	1.093	1.992	2.513
power	2.439	19.157	9.024	5.331	19.999	13.511
rectangle	10.718	1.601	12.275	21.921	2.341	23.043
runkut	0.148	0.449	0.226	0.082	0.890	0.265
sendmm	31.449	11.531	45.934	21.223	11.164	33.713
sumlist	0.028	0.048	0.030	0.013	0.054	0.022
trap	6.258	0.538	7.512	10.721	0.793	10.089
vecmat1	0.262	0.600	0.416	0.205	0.968	0.417
vecmat2	0.542	0.874	1.519	0.711	1.087	2.061

Table 11.8: Timings (sec.) of the norm- and nonnorm-analysers for the norm-programs

Program	imprecise abstraction		improved abstraction	
	$\mathcal{M}$	$\mathcal{DM}$	$\mathcal{M}$	$\mathcal{DM}$
mortgage1	0.158	0.499	2.921	3.179
mortgage2	0.110	0.329	0.901	1.190
mortgage3	0.168	0.118	1.707	0.150
power	19.015	9.180	–	80.183
runkut	0.509	0.157	26.959	0.172
trap	0.872	4.595	1.362	5.442
vecmat1	0.790	0.329	0.614	0.291
vecmat2	0.887	1.231	1.029	1.349

Table 11.9: Timings (sec.) for nonnorm-programs involving non-linear constraints

non-linear constraints are considered; a comparison for norm-programs should be similar). The abstraction with improved precision may be up to an order of magnitude slower than the imprecise one; for the $\mathcal{M}$ analysis of *power*, it even means getting no result (within a reasonable memory and time consumption).

11.3 Accuracy results

The accuracy of the analysers is determined by comparing the outcome of concrete executions of the benchmarks with the results obtained by the analyses. More precisely, the

Program	TotAnnot	ImpD	ImpF	PrecD	PrecF	PrecD+F
dnf	3772	0	33	100.0	99.1	99.1
fib	12	0	0	100.0	100.0	100.0
laplace1	112	0	0	100.0	100.0	100.0
laplace2	124	0	60	100.0	51.6	51.6
listlength	12	0	0	100.0	100.0	100.0
meal	56	0	0	100.0	100.0	100.0
mining	1064	105	80	90.1	92.5	82.6
mortgage1	54	0	8	100.0	85.0	85.0
mortgage2	36	0	0	100.0	100.0	100.0
mortgage3	36	3	2	91.6	94.4	86.0
num	1402	0	0	100.0	100.0	100.0
power	1256	126	73	89.9	94.2	84.1
rectangle	343	0	4	100.0	98.8	98.8
runkut	119	0	14	100.0	88.2	88.2
sendmm	141	0	0	100.0	100.0	100.0
sumlist	12	0	0	100.0	100.0	100.0
trap	135	73	8	46.0	94.0	40.0
vecmat1	125	0	6	100.0	95.2	95.2
vecmat2	208	0	13	100.0	93.7	93.7
Average				95.7	94.0	89.7

Table 11.10: Accuracy of the nonnorm-analysers for nonnorm-programs

Program	TotAnnot	ImpD	ImpF	PrecD	PrecF	PrecD+F
mortgage1	54	0	2	100.0	96.3	96.3
mortgage2	36	0	0	100.0	100.0	100.0
mortgage3	36	3	2	91.6	94.4	86.0
power	1256	126	55	89.9	95.6	85.6
runkut	119	0	4	100.0	96.6	96.6
trap	135	73	3	46.0	97.8	43.7
vecmat1	125	0	0	100.0	100.0	100.0
vecmat2	208	0	11	100.0	94.7	94.7
Average				90.9	96.6	87.8
OldAverage				90.9	93.1	84.0

Table 11.11: Accuracy with improved abstraction of non-linear constraints

correct (concrete) modes of the variables at each program point are compared with the modes derived by the analysers. If specialised versions of a predicate arise during concrete execution, these are considered separately; the predicate versions resulting from the analyses are mapped onto the concrete versions (usually, there is a one-to-one correspondence between the concrete and abstract predicate versions; in some cases however, several abstract versions map onto one concrete version or vice versa). We only checked the accuracy of the non-normalised analyses. For the normalised ones, the results should be worse as normalisation has a negative effect on the precision (cf. Section 9.1). The figures are pre-

sented in Table 11.10. Column "TotAnnot" gives the total number of variable annotations (summed up over the predicate versions and the program points), "ImpD" and "ImpF" give the number of imprecise variable annotations (derivation of mode a instead of d, resp. mode a instead of f). The columns "PrecD" and "PrecF" give the percentages of variable modes that are correctly inferred resp. by the $\mathcal{D}$ analysis and the $\mathcal{M}$ analysis; "PrecD+F" is the percentage of correct variable modes derived by the combined $\mathcal{DM}$ analysis. The average precision is shown at the bottom line. For the $\mathcal{D}$ and $\mathcal{DM}$ analyses, the worst case occurs for the *trap* benchmark (resp. 46% and 40%); for the $\mathcal{M}$ analysis, the worst results are for the *laplace*2 benchmark (51.6%).
The results in Table 11.10 are obtained by the analysers that abstract non-linear constraints in an imprecise way (at least as far as the freeness part is concerned). With the improved abstraction of Section 9.3, the average precision for the programs involving non-linear constraints improves from 93.1% to 96.9% for $\mathcal{M}$ and from 84% to 87.8% for $\mathcal{DM}$ (cf. bottom lines in Table 11.11; "Average" is the average precision of the improved abstraction, "OldAverage" is the average of the imprecise abstraction). For the *vecmat*1 benchmark the improved analysis even yields precise mode annotations.

Besides the accuracy of mode annotations, one can additionally consider the accuracy of the dependency information[1]. Even if correct modes are inferred at a particular program point, the inferred possible dependencies may not occur in the concrete case or may be too strong compared to the concrete dependencies, thus possibly leading to imprecise mode annotations at subsequent program points. Imprecise dependency information in addition to the imprecise mode information is derived when analysing the *sendmm*, *power* and *runkut* benchmarks.

There are three sources of inaccuracy : (1) the lack of information about term structures, (2) the treatment of non-linear constraints and (3) the abstraction of primitive constraints instead of the abstraction of conjunctions of primitive constraints. When selecting a component of a partially instantiated term having mode a, the definiteness analysis cannot discover the definiteness of a definite subterm; similarly, the freeness analysis cannot recognise free variables within the term. Avoiding normalisation of the input programs already improves the precision (cf. Section 9.1). For example, consider a call $p(f(X, Y))$ where X and Y are free before executing the call; matching this call with a clause head $p(f(A, B))$ correctly derives mode f as the mode for A and B. However, consider a program scheme of the form

$$\mathit{build\_structure}(\mathit{Data}), \mathit{constrain}(\mathit{Data}), \mathit{instantiate}(\mathit{Date})$$

where one first builds up a data-structure, then imposes constraints on that structure and finally instantiates it. Such a scheme is used quite frequently within CLP (e.g. *mining*, *power*, *rectangle*, *sendmm*,...). It gives rise to a loss of precision when selecting components of the structure within *constrain*/1 and *instantiate*/1. Imprecision due to the absence of structure information occurs in case of the *dnf*, *rectangle*, *laplace*2 and *mining* benchmarks, and is also causing part of the imprecision in *power* and *vecmat*2. Although adding structure information [59, 22] could clearly improve efficiency, it also complicates the analysis in the sense that, when changing the abstract representation for the unification part, also

[1] We only consider the *possible* dependency information; a similar study could be made for the definite dependencies.

the interaction between the unification and numerical part has to be revised. The second source of imprecision is due to the abstraction of non-linear constraints. This is the cause of inaccuracy in *runkut*, *trap*, *mortgage*1, *mortgage*3, *vecmat*1 and also partly in *power* and *vecmat*2. Finally, in the *sendmm* benchmark, imprecise possible dependency information is derived due to abstracting primitive constraints and joining their abstraction via abstract conjunction, instead of abstracting a conjunction of primitive constraints at once. In theory, loss of precision (at least for the freeness part) is also possible due to the imprecise abstraction of disequations and inequalities. However, it did not occur in the considered benchmarks. Also minimisation could lead to loss of precision, by combining dependencies (via union) that are unrelated (i.e. result from different OR-branches in the computation). Note that this is not as bad as applying transitivity on dependency relations, as is done for some LP mode analyses (e.g. the one of Musumbu [93]), but it may nevertheless lead to imprecise results. Again, no such imprecision was found for the benchmarks (in "real" programs, different predicate clauses usually establish dependencies between the same sets of variables).

11.4 Use of the information

Global program analysis has proven to be useful in the compilation and transformation of LP (Prolog) programs [110, 107]. In the case of CLP, the opportunities for optimisation are even larger. This is due to the fact that constraint solving is more involved than unification. Both low-level and high-level optimisations can considerably improve performance. These optimisations have already been explained in Chapter 4. In this section, we illustrate which of these optimisations can be based on the information derived by the analyses presented in the preceding chapters. An advantage of combining the definiteness and freeness analyses, besides an efficiency improvement of the freeness analysis, is that full mode information is available : modes f, d and a are derived, whereas the definiteness analysis derives only modes d and a and the freeness analysis modes f and a. Some optimisations only require this full mode information, others are based on the possible dependency information.

11.4.1 Constraint specialisation

A low-level optimisation is constraint specialisation. As explained in Chapter 4, by the time a constraint is encountered during execution, it is often a simple Boolean test or an assignment. A call to the solver can then be replaced by the appropriate test or assignment. This both decreases the size of the constraint store and the execution time. Most of the current CLP systems already avoid the invocation of the constraint solver in case the constraint can be specialised; however, checking whether specialisation is possible is still performed at run-time. Global compile-time analysis and specialised compilation can eliminate this overhead.

Constraint specialisation is based on mode information. Concerning freeness, the more restrictive notion of freeness is needed, i.e. a variable is free if it is unconstrained except for aliasing (binding) with other free variables (mode f_u of Section 9.5). A numerical equation can be transformed into an assignment if all but one of its variables are definite and the remaining one is free. A numerical disequation or inequality turns into a simple test if all

of its variables are definite. Even if only part of the variables in a numerical constraint are known to be definite, this information can be used to evaluate the definite subexpression(s) and substantially reduce the complexity of the constraint solving.

Example 11.4.1
Consider the mortgage program.

$$
\begin{aligned}
&mortgage(P, T, I, B, MP) \leftarrow \\
&\qquad T = 1, \\
&\qquad B = P * (1 + I) - MP. \\
&mortgage(P, T, I, B, MP) \leftarrow \\
&\qquad T > 1, \\
&\qquad T1 = T - 1, \\
&\qquad P1 = P * (1 + I) - MP, \\
&\qquad mortgage(P1, T1, I, B, MP).
\end{aligned}
$$

If the second argument in an entry pattern for mortgage is definite, then the second argument is inferred to be definite in all subsequent calls. This implies that $T = 1$ and $T > 1$ become simple tests. The analysis also derives that $T1$ is free before considering the constraint $T1 = T - 1$. So the equation can be transformed into an assignment to $T1$. If the third argument I is definite in the entry pattern, then I will remain definite in all subsequent calls. Hence the subexpression $1 + I$ can simply be evaluated and used as the coefficient of P in the equation.

11.4.2 Using specialised constraint solver instructions

Mode information can not only lead to avoiding the invocation of the constraint solver, but it can also give rise to the design and use of specialised constraint solver instructions. In [55, 56], the design of an abstract machine instruction set for a CLP system is addressed. We follow this description and slightly generalise one of the special cases of (numerical) constraint solving. The discussion is mainly focussed onto the treatment of linear equations (for the treatment of other constraints we refer to [56]).

First we introduce some important notions. A linear parametric form is an expression $n_0 + n_1X_1 + \ldots + n_kX_k$ where each n_i is a number and each X_i is a distinct solver variable. A linear equation is stored in the form $V = n_0 + n_1W_1 + \ldots + n_kW_k$ where V is a non-parameter and the W_i are parameters.
When encountering a linear equation during execution, first of all a parametric form is built into an accumulator (register or stack location). In order to do this there are instructions of the form `initpf` n and `addpf` n, X where n is a number and X is a variable. The former initialises a parametric form containing just the number n and puts it into an accumulator. The latter adds a term of the form $n * pf(X)$ to the parametric form being stored in the accumulator, where $pf(X)$ is the parametric form for X in the store (note : if X is known to be definite, a specialised form `addpf_def` n, X could be used). Thus the accumulator in general stores an expression exp of the form $n + n_1X_1 + \ldots + n_kX_k$. Afterwards, the instruction `solve_eq0`[2] tests for consistency (satisfiability) of $exp = 0$ in conjunction with

[2]There are similar instructions for inequalities.

the store. If consistent, the solver adds the equation to the store; otherwise, backtracking occurs. The process of solving an equation built using a parametric form roughly amounts to (a) finding a parameter V in the form to become non-parametric, (b) writing the equation $exp = 0$ into the form $V = pf$, (c) substituting out V using pf in all other constraints, and finally (d) adding the new constraint $V = pf$ to the solver.

Important special kinds of constraints justify making specialised versions of the basic `solve` instruction. One case is the following : the constraint has to be added to the store, but no satisfiability check is required. Often, the constraint can be organised into a form such that its consistency with the store is obvious. This typically happens when a variable with mode $\mathbf{f}_u$ appears in an equation[3]. This leads to the introduction of a specialised instruction `solve_no_fail_eq X` which adds the equation $X = exp$ to the store (exp is the expression in the accumulator). The main difference with `solve_eq0` is that no satisfiability check is performed.

Example 11.4.2
Consider adding the constraint $X + Y = 5$ to the store. Suppose that $Y = 3 + Z$ is already in the store and that Y and Z have mode $\mathbf{f}$ (i.e. they are constrained to be numerical but can still take all possible numerical values); also assume that X has mode $\mathbf{f}_u$ before adding $X + Y = 5$ (X is unconstrained except for possible aliasing with other variables of mode $\mathbf{f}_u$, so X is not yet involved in the numerical part of the store). Let X' be the variable obtained after dereferencing X (i.e. the variable at the end of the reference chain for X). A direct compilation results in the following code (the rightmost column depicts the current state of the accumulator) :

```
initpf    5          accumulator : 5
addpf     -1, X'     accumulator : 5 - X'
addpf     -1, Y      accumulator : 2 - X' - Z  (note: Y = 3 + Z)
solve_eq0            solve : 2 - X' - Z = 0
```

A better compilation can be obtained by using the specialised instruction `solve_no_fail_eq X'`. *This yields :*

```
initpf          5          accumulator : 5
addpf           -1, Y      accumulator : 2 - Z
solve_no_fail_eq X'        add : X' = 2 - Z
```

Other special cases of constraint solving described in [55, 56] are based on the notions of future-redundancy and dead variable removal (for the latter cf. Section 11.4.8). The detection of these cases requires sophisticated analyses and are not possible using only the analyses that we have presented.

[3] In [55], a stronger condition is applied : only *new* variables, i.e. completely unconstrained variables, are considered.

11.4.3 Detection of linearity

In most of the CLP systems, non-linear numerical constraints are delayed until they become linear through the definiteness of certain variables. If it is known at compile-time that a non-linear constraint will always be linear when encountered during execution, the compiler does not need to produce the code for dealing with the more complex case of non-linearity. The detection of linearity is based on definiteness information.

Example 11.4.3
Consider again the mortgage program. If I or P are known to be definite each time the equations $P1 = P*(1+I)-MP$ or $B = P*(1+I)-MP$ are encountered (as is the case for the entry pattern mortgage(f, d, d, d, d) for example), then these equations turn into linear ones.

11.4.4 Mutual exclusion

By the time an atom is resolved, it often happens that only one of the clauses defining the atom has constraints consistent with the current store. In other words, the clauses defining a predicate are *mutually exclusive* for a particular entry pattern that indicates the definiteness of some of the arguments. If an argument is free (mode f_u), indexing on that argument must be avoided. So mode information is needed for this optimisation. In case of mutual exclusion, there is no need to set up a choicepoint, as subsequent backtracking to the predicate will lead to failure.

Example 11.4.4
In the mortgage program, the constraint $T = 1$ in the first clause and $T > 1$ in the second clause cannot both be satisfiable if T is bound to a specific value. Hence for entry patterns in which T is definite (which implies the definiteness of T in the following recursive calls), the mortgage predicate can be compiled into the following deterministic pseudo target code:

$$
\begin{array}{l}
mortgage(P, T, I, B, MP) \leftarrow \\
\quad \textit{if } T = 1 \textit{ then} \ldots \\
\quad \textit{else if } T > 1 \textit{ then} \ldots mortgage(\ldots) \textit{ else fail}.
\end{array}
$$

11.4.5 Reordering of primitive constraints

The mode information can also be used to reorder primitive constraints. The idea is to detect failure as soon as possible by putting the most "constraining" constraints first. For primitive numerical constraints a simple ordering rule can be the following : a primitive numerical constraint is more constraining than another if the number of its free variables is smaller or, in case this number is equal, if it contains more non-free variables. The notion of freeness used here is the less restrictive one, i.e. a variable is free if it is unconstrained except for possible type-constrainedness or for possible aliasing with other free variables (mode f).

Example 11.4.5
Consider the program fragment

$$\ldots, (P)\ X + Y = Z, Y - 1 = T, \ldots$$

where X and Y are free and Z and T are definite at program point (P). Then $Y - 1 = T$, which contains 1 free variable, is more constraining than $X + Y = Z$ with two free variables. Hence the two constraints should be swapped. By reordering the constraints, Y will have become definite due to the execution of $Y - 1 = T$ such that $X + Y = Z$ can then be transformed into an assignment to X.
Another example is taken from the num benchmark :

$$moinsCent(N, W, P) :- U > 0, N = 20 + U, \ldots$$

*where the entry pattern of moinsCent is moinsCent(**d**, **f**, **f**). Then $N = 20 + U$ (with one definite and one free variable) is more constraining than $U > 0$ (with one free variable). Hence $U > 0$ should be put after $N = 20 + U$.*

However the simple ordering rule given above is not always adequate. Especially when also primitive unification constraints are involved, deriving the constraint order becomes more complex. One definite/non-free variable in a unification constraint may cause the definiteness/non-freeness of a set of other (free) variables, and therefore the constraint may be quite constraining; e.g. $X = f(Y_1, Y_2, Y_3, Y_4)$ with X being definite and all Y_i being free will cause all Y_i to become definite when executed. Designing rules for primitive constraint ordering is outside the scope of this thesis and is a subject for further research. In any case, mode information is indispensable in this context.

11.4.6 Independence

Definite/possible failure information (cf. Section 9.4) forms the basis for many optimisations of CLP programs. Failure information captures (possible) interaction (also called dependence) between constraints and/or goals in a program. Intuitively, two literals p and q are *independent* with respect to a store C if p and q do not "affect" each other in the sense that the constraints added to the store C by p and q do not interfere, i.e. do not cause failure. This is called *search independence* in [44].

One application of search independence consists of reordering a (sub)goal "C, p" to "p, C", where C is a constraint and p is an atom, if C and p are independent. The motivation for this reordering is that variables in C may become definite due to p, enabling C to be translated into either an assignment or a simple test. Especially in the case p is recursive, large speedups are obtained [63, 86]. The criterion for safe constraint-goal reordering is given by the following proposition.

Proposition 11.4.1 (Criterion for constraint-goal reordering (weak))
Consider the (sub)goal "$(AC_1)\ C,\ (AC_2)\ p\ (AC_3)$" where $C \in Cons$, $p \in Atom$ and the AC_i are the abstract constraints[4] at the relevant program points.
Then "C, p" can safely be reordered to "p, C" iff

[4]Unless explicitly mentioned, "abstract constraint" is used as an abbreviation of "compound abstract constraint" in the sequel; to be precise, we should write AC_i^t instead of AC_i.

1. $AC_1 \notin \{\bot, Afail\}$;
2. $AC_2 \neq Afail$ *and* $\emptyset \notin AC_2$;
3. $AC_3 \neq \bot$ *and* $\emptyset \notin AC_3$.

Note that $AC_2 \neq \bot$ if $AC_1 \notin \{\bot, Afail\}$; also, $AC_3 \neq Afail$ (Proposition 9.4.1). If AC_1 is $\bot$ or *Afail*, then C, p cannot be reached; so reordering is irrelevant. If AC_2 does not satisfy condition 2, it means that either the constraint C is equivalent to *false* or C may interfere with its call store (abstracted by AC_1) and lead to possible failure, in both cases causing pruning. Hence C should not be shifted over p. If AC_3 contains $\emptyset$, failure may occur during execution of p due to interference of the constraints represented by AC_2 (including C) with the local constraints of p (Proposition 9.4.1). Hence C should not be shifted over p. If $AC_3 = \bot$ and assuming that $AC_2 \neq Afail$, $\bot$ is due to local unreachability or failure without interference of the call environment of p (i.e. without interference of the constraints represented by AC_2). However, this local information may overrule possible failure that is due to interaction between the constraints in p and AC_2 and that occurs at a previous program point, as illustrated by the program fragment below (possible failure detected after $X = 3$ due to the interference with $X = 1$ will be overruled by the definite failure of $1 = 2$) :

$$\ldots, X = 1, p(X), \ldots$$
$$p(X) \leftarrow X = 3, \ldots, 1 = 2, \ldots$$

Hence, it is not safe to shift C over p since C might interfere with p.

Unfortunately, the above criterion misses out opportunities for constraint shifting. The reason is that possible failure in AC_3 may be caused by interference of the constraints in p with constraints in AC_2 that have nothing to do with C. The latter are constraints that are already represented by AC_1 and just passed on to AC_2. The following example illustrates the problem.

Example 11.4.6
Consider the sumlist program called with the first argument possibly non-free and the second argument free. The aim is to move the constraint $S = H + S1$ after the recursive call to sumlist.

$$sumlist(L, S) : -L = [\,], S = 0.$$
$$sumlist(L, S) \leftarrow L = [H \mid T],\ (AC_1)\ S = H + S1,\ (AC_2)\ sumlist(T, S1)\ (AC_3).$$

AC_1 contains $\{T\}$ which represents a constraint on T that is obtained by the conjunction of $L = [H \mid T]$ with the call store constraining L; $\{T\}$ is just passed on to AC_2 during abstract interpretation of $S = H + S1$. The recursive call to sumlist also constrains T (T will be bound to a list). This local constraint may interfere with the constraint on T represented by AC_2, leading to the indication of possible failure in AC_3 ($\emptyset \in AC_3$). Hence, according to the above reordering criterion, $S = H + S1$ may not be shifted over $sumlist(T, S1)$, although this constraint is not involved in the possible failure at AC_3 !

To solve this problem, one should only consider that part of AC_2 in which C is involved. Considering $AC_2 \setminus AC_1$ and the way in which this set interferes with the local constraints

in p is not safe : $AC_2 \setminus AC_1$ does not capture the information that was already in AC_1 but is once more added by C or by the combination of C with information in AC_1. This is shown by the following example.

Example 11.4.7
$q(X,Y),(AC_1)\ X+Y=3,\ (AC_2)\ ,p(X),\ (AC_3),\ldots$
$q(X,Y) \leftarrow X=1.$
$q(X,Y) \leftarrow Y=1.$
$p(X) \leftarrow X=3,\ldots.$

Assume that the store before analysing $q(X,Y)$ is empty, so after $q(X,Y)$ we obtain $AC_1 = \{\{X\},\{Y\}\}$. Then $AC_2 = AC_1 \wedge^{\mathcal{F}} \alpha^{\mathcal{F}}(X+Y=3) = \{\{X\},\{Y\},\{X,Y\}\}$; hereby the sets $\{X\}$ and $\{Y\}$ that were already in AC_1 and are passed on to AC_2 are added once more via the combination of $X+Y=3$ with resp. $Y=1$ and $X=1$ which yields resp. $X=2$ and $Y=2$. $AC_2 \setminus AC_1 = \{X,Y\}$, so any local constraint on X added by p (abstracted as $\{X\}$) cannot cause possible failure when joined with $AC_2 \setminus AC_1$. However, the combination of C with the information in AC_1 does influence the behaviour of p : the combination of $X+Y=3$ with $Y=1$ entails $X=2$ which is incompatible with $X=3$ within p. This implies that C should not be shifted over p !

The only correct way to obtain the part of AC_2 in which C is involved is to compute $\alpha(C) \cup (no\_fail(AC_1) \oplus \alpha(C))$[5]. Note that there is no way to extract this information out of the given annotations AC_1 and AC_2. The interference of this information with the local constraints in p has to be computed; this is done by joining the information with AC^n_{exit} representing the constraints added during execution of p. So the criterion for constraint shifting based on this approach requires some extra computation in exchange for the extra precision.

Proposition 11.4.2 (Criterion for constraint-goal reordering (strong))
Consider the (sub)goal "$(AC_1)\ C,\ (AC_2)\ p\ (AC_3)$" where $C \in Cons$, $p \in Atom$ and the AC_i are the abstract constraints at the relevant program points.
Then "C,p" can safely be reordered to "p,C" iff

1. *$AC_1 \notin \{\bot, Afail\}$;*
2. *$AC_2 \neq Afail$ and $\emptyset \notin AC_2$;*
3. *$AC_3 \neq \bot$ and $\emptyset \notin AC'_3$ where $AC'_3 = no\_fail(\alpha(C) \cup (no\_fail(AC_1) \oplus \alpha(C))) \wedge no\_fail(AC^n_{exit})$[6] and AC^n_{exit} is the least upper bound of the n components of the abstract constraints at the end of the clauses defining p.*

Note that AC'_3 is a subset of AC_3. With this improved criterion, it is possible to derive the safe reordering of "$(AC_1)\ S = H + S1,\ (AC_2)\ sumlist(T,S1)\ (AC_3)$" to "$sumlist(T,S1),\ S = H + S1$" for the sumlist example (more precisely, with the analysis in Section 9.4 we have that $AC_1 = close(\{\emptyset,\{L\},\{H\},\{T\}\})$, $AC_2 = close(\{\{L\},\{H\},\{T\},\{S,S1\}\})$, $AC'_3 = no\_fail(close(\{\{S,S1\},\{S,S1,H\},\{S,S1,L\},\{S,S1,T\}\})) \wedge no\_fail(close(\{\{T\},\{S1\}\})) = close(\{\{H,S\},\{L,S\},\{S\},\{S1\},\{T\}\})$).

[5]The operations α, $\oplus$, $\wedge$ and *no_fail* used here and in the rest of this section are the ones defined in Section 9.4.

[6]The *no_fail* operations ensure that possible failure discovered in AC'_3 can only be due to the *combination* of the local constraints in p with the call constraints involving C.

A sufficient (but not necessary) condition which is easier to check and which is adequate for a lot of situations in practice (e.g. for the *sumlist* program) can be derived from the above criterion :

Proposition 11.4.3 (Sufficient criterion for constraint-goal reordering)
Consider the (sub)goal "(AC_1) C, (AC_2) p (AC_3)" where $C \in Cons$, $p \in Atom$ and the AC_i are the abstract constraints at the relevant program points.
Then "C, p" can safely be reordered to "p, C" iff

1. $AC_1 \notin \{\bot, Afail\}$;
2. $AC_2 \neq Afail$ *and* $\emptyset \notin AC_2$;
3. $AC_3 \neq \bot$ *and* $\overline{\exists}_{vars(p)} no\_fail(\alpha(C) \cup (no\_fail(AC_1) \oplus \alpha(C))) = \emptyset$.

The third condition implies that no dependency established by C or by the combination of C with the constraints in its call store involves only the variables of the call p; hence C cannot influence the execution of p and therefore C can safely be put after p. Or, stated in another way, if the third condition is fulfilled, then the third condition of Proposition 11.4.2 is implied. The proof is by refutation. Given that the third condition in Proposition 11.4.3 holds, assume that the third condition in Proposition 11.4.2 is not implied, i.e. $\emptyset \in AC_3'$. Using the definitions of *no_fail* and $\wedge$, $\emptyset \in AC_3'$ can only be true if $\emptyset \in no\_fail(\alpha(C) \cup (no\_fail(AC_1) \oplus \alpha(C))) \oplus no\_fail(AC^n_{exit})$. By definition of $\oplus$, this implies that there should exist a $S_1 \in no\_fail(\alpha(C) \cup (no\_fail(AC_1) \oplus \alpha(C)))$ and a $S_2 \in no\_fail(AC^n_{exit})$ such that $S_1 = S_2$ (then $(S_1 \cup S_2) \setminus D = \emptyset$ if $D = S_1 \cap S_2$). However, $\overline{\exists}_{vars(p)} no\_fail(\alpha(C) \cup (no\_fail(AC_1) \oplus \alpha(C))) = \emptyset$ and $vars(no\_fail(AC^n_{exit})) \subseteq vars(p)$. So $S_1 \not\subseteq vars(p)$ whereas $S_2 \subseteq vars(p)$, hence S_1 cannot be equal to S_2. As a consequence, the assumption $\emptyset \in AC_3'$ cannot hold.
Note that reordering "(AC_1) C, (AC_2) p (AC_3)" to "(AC_1) p, (AC_2') C (AC_3')" in general affects the abstract information at program points reached during execution of p: after reordering, the abstraction at those points does no longer include the abstraction of C or combinations of it with the local procedure constraints (since the analysis of p then starts with AC_1 rather than AC_2). However, there is a special case in which no recomputation is needed, namely if $\overline{\exists}^{\mathcal{F}}_{vars(p)} no\_fail(AC_1) = \overline{\exists}^{\mathcal{F}}_{vars(p)} no\_fail(AC_2)$. The sumlist example satisfies this condition : $AC_1 = close(\{\emptyset, \{L\}, \{H\}, \{T\}\})$, $AC_2 = close(\{\{L\}, \{H\}, \{T\}, \{S, S1\}\})$, so $\overline{\exists}^{\mathcal{F}}_{\{T,S_1\}} no\_fail(AC_1) = \overline{\exists}^{\mathcal{F}}_{\{T,S_1\}} no\_fail(AC_2) = \{\{T\}\}$ (note : $\{T, S_1\} = vars(sumlist(T, S_1))$). In fact, we believe that in many practical situations in which safe reordering is derived, this condition holds so that recomputation of the abstract information within p is avoided.
Another piece of information that still has to be computed after reordering is the abstract success constraint AC_2' of p (which is also the abstract call constraint of C) : $AC_2' = no\_fail(AC_1) \wedge no\_fail(AC^n_{exit})$ where AC^n_{exit} is the abstract information that was added during execution of p (local information). The abstract constraint AC_3' can only differ from AC_3 in the sense that it no longer includes $\emptyset$, so AC_3' can be computed as $AC_3' = AC_3 \setminus \{\emptyset\}$. The fact that $\emptyset$ cannot be in AC_3' is implied by the reordering conditions 2 and 3: C itself or the combination of C with the constraints so far (now also including the constraints gathered during execution of p) cannot cause failure; otherwise it would not have been safe to perform the reordering.
The reordering criteria above are formulated in terms of the $\mathcal{F}$ abstraction. They also hold in terms of the $\mathcal{M}$ abstraction. When dealing with the $\mathcal{DF}$ or $\mathcal{DM}$ abstraction, the

corresponding $\mathcal{F}$ or $\mathcal{M}$ information can be obtained by extending the abstract constraints using the *extend* or *extend*^{m} operation (cf. Definitions 7.2.3 and 8.2.3).

The reordering can be generalised to reordering of arbitrary literals p and q. In this case, the reordering of "p, q" to "q, p" is allowed if changing the order (1) will not increase the search space and (2) will not increase the overall execution time. As pointed out in [44], it is difficult to give simple and general conditions that ensure the above requirements. However, a sufficient set of conditions is that (1) p and q are search independent with respect to the call store C of "p, q", (2) q is single-solution and (3) p has at least one answer when starting its computation from C (for more details we refer to [44]). So search independence on itself is no longer sufficient.

A second application of independence information is AND-parallelisation. The idea is to execute two literals p and q in parallel if this allows to obtain the same answers as those obtained by sequential execution, but in a shorter time. This leads to the following requirements : the literals p and q in a (sub)goal "p, q" can be executed in parallel if they are search independent and if the constraint solver is weakly independent. Intuitively, the latter means that changing the order in which constraints that do not interfere are added to the constraint solver, gives rise to only a "small" change in execution time. In [44] it is shown that the constraint solvers for CLP(H,N) are weakly independent, hence the above requirements are reduced to search independence.

Other applications of independence information are compile-time stability detection in the Andorra family of languages [58] and intelligent backtracking [44].

11.4.7 Dead code removal

Information about definite failure allows to remove dead code. More precisely, the literals in a clause after a program point annotated with $\bot$ (or *Afail* in case of the analysis described in Section 9.4) can be deleted. If the first program point in a clause is annotated with $\bot$ or *Afail* then the entire clause can be removed.

Example 11.4.8
For the non-normalised dnf benchmark with entry pattern $dnf(\mathbf{d}, \mathbf{f})$, *the DM analysis gives rise to a number of specialised predicate versions. An example of such a version for the dnf predicate is one with call pattern* $dnf(o(n(\mathbf{d}), n(\mathbf{d})), \mathbf{a})$. *For almost all of these versions, some of the clauses for the dnf predicate can be deleted. For example for the version above the third clause defining dnf has the form*

$$dnf(a(X, Y), a(X, Y)) \leftarrow literal(X), literal(Y).$$

Since the head of this clause does not match with the call pattern of the specialised version above, the clause cannot be used to resolve the call and can be deleted; the analyser indicates this by deriving $\bot$ *for the abstract constraint at the beginning of the clause.*

Another opportunity for code removal is the deletion of future-redundant constraints. Recall that a future-redundant constraint c does not have to be added to the constraint store, but in general still has to be checked for consistency with that store. However, if the failure analysis (Section 9.4) derives that the abstract success constraint of c is different from *Afail*

and does not contain $\emptyset$ (i.e. neither definite nor possible failure occurs), the consistency is always ensured and the future-redundant constraint c can be removed completely.

Example 11.4.9
Consider the listlength program.

$$listlength([\,],0).$$
$$listlength([H \mid T],N) \leftarrow N \geq 1, N = N1 + 1, listlength(T,N1).$$

Suppose that the entry is $listlength(\mathrm{a},\mathrm{f})$. *The constraint* $N \geq 1$ *in the second clause is future-redundant (*$N \geq 1$ *is entailed by* $N1 \geq 0 \wedge N = N1 + 1$*). With the given entry pattern, the abstract success constraint of* $N \geq 1$ *is (applying minimisation)* $\{\{H\},\{T\},\{N\}\}$. *This abstract constraint indicates definite satisfiability at that point. Hence* $N \geq 1$ *can be removed when generating the specialised version of the listlength program for the given entry pattern, thus avoiding the invocation of the simplex solver for inequalities.*

11.4.8 Dead variable removal

Typically in the execution of a CLP($\mathcal{R}$) program, many variables remain in a state where the constraint solver continually manipulates them, even if they will never be referenced in the remaining computation. These variables are called *dead* variables [75].

Example 11.4.10
Consider again the sumlist program. The execution of the goal $sumlist([1*V, 2*V, \ldots, n*V], S)$ *subsequently sets up the constraints*

$$S = 1*V + S_1, \quad S_1 = 2*V + S_2, \quad \ldots, \quad S_n = 0.$$

Clearly, after the second constraint is added, the variable S_1 *will never be referred to again in the execution. By projecting this variable out of the first two constraints to obtain* $S = 3*V + S_2$, *no information is lost but the number of constraints in the store decreases. A similar reasoning applies to* $S_2, \ldots, S_n$. *By removing these variables, the constraint store will eventually contain only one constraint instead of n constraints involving n local variables that all need to be manipulated when* V *is further constrained.*

The removal of *dead* variables requires a fairly sophisticated program analysis. Such an analysis is described in [75]. An abstract constraint is a tuple (CA, SH, GR, NL) :

- CA : this set contains the so-called *call alive* variables which are those passed in from an upper level call. They may be accessed after the current call terminates.
- SH : a set of unordered pairs of variables which may share, i.e. the usual possible sharing information derived only from the unification constraints, not from the numerical constraints (e.g. cf. [15]). A special element is added to express sharing with variables outside the scope of the current clause.
- GR : a set of groundness (i.e. definiteness) dependencies. These dependencies are represented by formulae of the PROP domain [81, 84].

- *NL* : non-linear information, which is held in formulae similar to the groundness dependencies (each non-linear dependency involves a possibly delayed variable and a groundness formula that will definitely wake up all non-linear constraints involving the variable).

The paper [75] contains a description and some examples of this analysis but does not report on an implementation.

Parts of the information needed in the dead variable analysis could be supplied by the $\mathcal{DF}$ (or $\mathcal{DM}$) analyser. More precisely, the definiteness dependencies GR are present in the definiteness part of the analysis and the sharing information SH is a subset of the possible dependency information in the freeness part (except that possible sharing with variables outside the scope of the current clause is not captured, but this could easily be added). The problem with the sharing information is to extract the required subset. This is only possible if type information is present (cf. Section 9.5) : the subset of dependencies that must be considered are the pairs $\{X, Y\}$ where X and Y are not both of type *numerical*.

11.4.9 Example

By means of example, we show how several of the above optimisations are applied to one of the simple benchmarks, being the *sumlist* program. The considered entry pattern is *sumlist*(**d**,**f**).

$sumlist(L, S) \leftarrow L = [\,], S = 0.$
$sumlist(L, S) \leftarrow L = [H \mid T], S = H + S1, sumlist(T, S1).$

The $\mathcal{DM}$ analysis derives that for all recursive calls the first argument T is definite and the second argument $S1$ has mode **f**. As a consequence,

- the first argument that is always definite can be used for indexing ([ ] and $[H \mid T]$ imply mutual exclusion);
- the constraint $L = [\,]$ is just a simple test, $S = 0$ can be transformed into an assignment to S (at least if S is known to be untyped) and $L = [H \mid T]$ is a selection of the head H and the tail T of the list L;
- the constraint $S = H + S1$ can be put after the recursive call as it cannot influence the executional behaviour of that call (cf. Section 11.4.6); after the recursive call, $S1$ will have become definite and together with the fact that H was already definite and S is free, $S = H + S1$ can be turned into an assignment to S.

Hence, the optimised code should be :

$sumlist(L, S) \leftarrow$
if $L = [\,]$ then $S := 0$
else if $L = [H \mid T]$ then $sumlist(T, S1); S := H + S1$
else *fail*.

Chapter 12

Conclusion

This thesis addresses the global analysis of constraint logic programs, formalised in terms of abstract interpretation. The aim is to derive run-time properties that can be used at compile-time to generate more efficient target code for a given set of queries. This optimising compilation is intended to overcome the efficiency problems of the current Constraint Logic Programming (CLP) systems, which often use general constraint solving where it is not really needed.

The technique of abstract interpretation has been extensively studied in the context of Logic Programming (LP). It was described in [43, 41] how one of the traditional frameworks for abstract interpretation of LP (Bruynooghe's framework [6]) can be extended to the analysis of CLP programs in a more or less straightforward way. The idea is to replace the notions of term domain, (abstract) substitution and (abstract) unification by constraint domain, (abstract) constraint and (abstract) conjunction of a constraint to the constraint store. The safety conditions are adjusted accordingly. A pair of lower-level abstract operations than the ones in the original framework can be identified, namely abstract conjunction and abstract projection, in terms of which the other operations can be defined. The framework provides a backbone for the development of applications, such as our freeness analysis: it specifies an application-independent algorithm and identifies some local requirements on the application-dependent components (abstract domain and abstract operations) in order to ensure the safety and termination of the overall analysis. Compared to other frameworks for abstract interpretation of (C)LP that have been proposed [83, 17, 46, 11], an additional advantage of using Bruynooghe's framework is that there exist efficient implementations of (variants of) it [94, 70].

Some important issues in designing a particular analysis of CLP programs can be characterised. These issues have also been pointed out by García de la Banda and Hermenegildo in [43]. A first point concerns the safe and accurate abstraction of constraint entailment. On the abstract level, this corresponds to maintaining information on entailed variable dependencies. For example, consider our freeness analysis. The interest is in determining whether a variable is definitely free, i.e. whether the variable can still take all possible values according to its type. This property could be abstracted by keeping track of the set of free variables. However, given this abstraction, almost all freeness information would be lost when applying abstract conjunction. The key point is that, no matter how this operation is defined, the abstraction does not contain enough information in order to mimic the propagation of non-freeness by the constraint solving process. Therefore, one has to

assume that all free variables become potentially non-free. This problem also occurred in early analysers of LP which did not (precisely) deal with variable sharing or aliasing [31]. In the context of CLP this corresponds to safely and accurately abstracting constraint entailment with respect to the target property. In other words, constraint entailment can be viewed as the CLP generalisation of the sharing or aliasing notion in LP : sharing or aliasing is established through equality constraints between Herbrand terms, whereas in the CLP case also other constraints lead to (entailed) variable dependencies. All (entailed) dependencies can be discovered by transforming constraints to solved form, as shown in Chapter 5 of this text. The level of accuracy in the abstraction and in the abstract operations, which is thus based on the particular abstraction of constraint entailment, determines the precision of the analysis but also its efficiency; the developer of the analysis then has to fix the desired trade-off between the two.

Developing abstract domains and corresponding abstract operations for the analysis of CLP programs is more complex than in the LP case. This is due to the intrinsic complexity of most constraint solving algorithms, which are more involved than the unification algorithm (consider for example the solved form algorithms for linear numerical constraints presented in [69, 52]). An important issue in this respect is what kind of language should be used to implement the analysis, or at least to implement the most involved abstract operation which is abstract conjunction. An interesting suggestion made by Codognet and Filé [17] (and also by Giacobazzi, Debray and Levi [46]) is to use a CLP language for that purpose. In that case, constraint solving algorithms come for free. The use of the constraint solving capabilities could greatly simplify the specification and implementation of the abstract algorithms.

Another substantial complication in the analysis of CLP programs is the treatment of interactions between several constraint domains. In general, there is no clear separation between objects of different domains. For example, the constraint $X = Y$ can belong to almost any constraint domain. Also, a constraint in one domain may entail a constraint in another domain. For example, the numerical constraint $2X - 2Y = 0$ entails a unification constraint $X = Y$, which may influence the constraint solving in the Herbrand domain; e.g. combining $X = Y$ with the unification constraints $W = f(X)$ and $V = g(Y)$ entails a dependency between W and V. Finally, some abstract information spans several constraint domains. For example, in our freeness analysis, the unification constraint $X = f(A, B)$ and the numerical constraint $A + B = T$ establish a dependency between X and T which is not related to one specific constraint domain. These observations imply that it is not sufficient to abstract the constraints and constraint solving algorithms for each constraint domain separately; also the combinations/interactions of domains must be taken into account. In our opinion, this is one of the most difficult parts in the design of a particular analysis and has not received enough attention in recent publications.

In this thesis, we presented a freeness analysis as an application developed within the extended abstract interpretation framework. This analysis aims at deriving information on possible constraint interaction or, stated in another way, on which variables act as degrees of freedom with respect to the satisfiability of the constraint store. First of all, a kernel abstraction was specified. This abstraction forms the theoretical basis and addresses only fundamental aspects. The emphasis is on the safety proof. Also, the analysis is focussed on CLP languages including the Herbrand domain (LP term domain) and a numerical domain (domain of real or rational numbers), although it can be extended to include other

constraint domains as well, as shown in Section 9.6.
Afterwards, we described two approaches for optimising the kernel abstraction. The first one consists of keeping track of only a minimum of information, rather than representing all information exhaustively. The compressed abstraction is a safe approximation of the original one. It allows to reconstruct a superset of the original abstraction. However, experiments have shown that for "real" programs usually no loss of precision is involved. The second approach exploits definiteness information which is for example supplied by the definiteness analysis of García de la Banda and Hermenegildo [43, 41]. It extracts the variable dependency information that involves definite variables from the original abstraction, and compresses it to the set of definite variables. An abstract constraint is thus split in two parts: one part containing the definite variables and another part containing information on the non-definite variables and their dependencies. This approach incurs no loss of precision. As both approaches are orthogonal, they can be combined yielding a practical full mode analysis system. To our knowledge, this is the first full mode analysis system for CLP. The advantage of the combined analysis is not only improved efficiency compared to running both the freeness and definiteness analysers separately. The derived information also forms the basis for a larger suite of program optimisations.

The analyses above (or at least their freeness parts) address only fundamental aspects. In order to increase their practicality and the precision and amount of derived information, we proposed a number of (orthogonal) extensions. First of all, input programs are no longer required to be normalised (i.e. arguments of calls and clause heads do not have to be distinct variables and constraints can be of any form). Another collection of extensions capture additional program properties, thereby possibly changing the form of the abstraction. These include a more precise treatment of disequations and inequalities and of passive constraints (for the numerical domain these are the non-linear constraints), the inference of definite and possible failure information and the addition of type information. The latter allows to distinguish between two notions of freeness : a restrictive one stating that a variable is free as long as it is unconstrained except for possible bindings with other free variables of that kind, and a less restrictive definition that considers a variable to be free if it can still take all possible values at least according to its type. Finally, it was shown how the analysis can be extended to handle other constraint domains, such as the PrologIII tuple domain.

A prototype implementation of the analysers (including some of the above extensions) has been developed within the abstract interpretation system `PLAI` [94, 96], which runs on top of SICStus Prolog. The system can be considered as a sophisticated implementation of Bruynooghe's framework, incorporating several optimisations to the abstract interpretation algorithm. The system was provided by the CLIP team at U.P.Madrid. They also generalised it to support the analysis of CLP programs and to allow the combination of their definiteness analysis with our freeness analyses that exploit definiteness information. The actual way of communication between the two analysers was studied in collaboration between U.P.Madrid and K.U.Leuven in the context of the PRINCE project.
The prototype has been applied to a collection of diverse CLP benchmarks. The examples given in Appendix A have been produced by the prototype. Evaluation of the accuracy and efficiency of the analysers has shown that the minimal freeness analysis combined with the definiteness analysis gives quite satisfactory results. Inaccuracy is mainly due to non-linear

constraints and to the absence of structure information. While the addition of structure information (cf. e.g. [59, 72]) could certainly improve precision, it also complicates the analysis, especially concerning the interaction between the constraint domains. This requires further investigation. Regarding efficiency, it appeared that running the definiteness and minimal freeness analysers in parallel sometimes caused a loss of efficiency due to the fact that one analyser forced the other into going along with more fixpoint iterations. An alternative could be to first let the definiteness analyser produce an annotated program, and afterwards apply the freeness analyser onto that program. Another factor influencing efficiency is the representation of the abstract constraints, more precisely of (sets of) sets of variables. The PLAI system represents a set of program variables as a set of free Prolog variables. An alternative could be to use a bit-like representation. It not only saves space, but is also expected to result in more efficient abstract operations. These are topics for further research. Further work also includes incorporating the described extensions that are still missing in the current prototype, extending the number of language built-in operations that can be dealt with, and evaluating the analysers on more and larger benchmarks.

The analysers developed in this thesis are tools for providing at compile-time information about run-time properties. This information allows to apply a large suite of program optimisations. Information on possible constraint interaction supplied by the freeness analysers (or the freeness part of the combined analysers) is needed to infer constraint/goal independence. This in turn gives rise to optimisations such as constraint/goal reordering, AND-parallelisation and intelligent backtracking. Definiteness information and the more restrictive kind of freeness information can be used for optimisations such as constraint specialisation or the use of specialised constraint solver instructions, the detection of linearity and the detection of mutual exclusion. Finally, definiteness information and information on possible dependencies between variables is also part of what is needed for dead variable removal [75].

The focus in this thesis has been on the derivation of correct program information. An important aspect that requires further study is the integration of the global analysers into a highly optimising CLP compiler. A first discussion on this topic can be found in [85]. The main issues mentioned there are (1) to detect which optimisations are applicable (note that even if an optimisation can be applied, it is not always guaranteed to produce a speedup or space saving because it may affect future constraint solving) and (2) to deal with non-trivial interactions between different optimisations (performing one optimisation may preclude another optimisation in another part of the program). Based on the preliminary studies of [85], a highly optimising compiler is expected to produce code that is at least an order of magnitude faster and more space efficient than existing implementations. This increase of efficiency, together with the expressive power of CLP languages, should lead to a more widespread acceptance and use of CLP.

Appendix A

Examples

The prototype implementation of the different analysers has been applied to a collection of CLP programs, described in Chapter 11. We give the analyser output for some of these programs. However, the larger programs are not presented here; the abstract interpretation system including the analysers and all benchmarks can be obtained upon request.

The output for a program is organised as follows :

- The heading gives the query pattern, consisting of a predicate call and its abstract entry constraint, and the corresponding abstract exit constraint.
- The analyser may derive several specialised versions for the predicates in the program (the first version has the original predicate name, the name of subsequent versions consist of the original predicate name followed by the version number). For each version, a header indicates the form of the concrete call(s) to this version and the abstract entry and exit constraint. Predicate calls are replaced by calls to the appropriate version. (Recall that the analyser offers the choice between this output form and a form containing only one general predicate version.)
- For each clause of a predicate version, each program point is annotated with an abstract constraint presenting the information that holds at that point. Although the analysis derives compound abstract constraints in order to limit the loss of precision, no distinction will be made here between the old and new components, since this is irrelevant for deriving modes and performing program optimisations. Instead, the union of the two components is computed (in minimal form).

The output is obtained using the $\mathcal{DM}$ analyser. This output allows to reconstruct (an approximation of) the output supplied by the other analysers. Let D be the set of definite variables and M^* be the set of possible variable dependencies, where singletons indicate possible non-freeness of the involved variables. The set D is the same in case of the $\mathcal{D}$ or $\mathcal{DF}$ analyser. Using D and M^*, one can construct what is derived by the $\mathcal{M}$ analyser (more precisely, for each variable in D, a singleton with that variable has to be added to M^* : $extend^m(D, M^*)$ as defined in Definition 8.2.3). In all programs considered, the extended information also corresponds to the minimised form of what is derived by the $\mathcal{F}$ or $\mathcal{DF}$ analyser; so no precision is lost by performing minimisation at each step during the $\mathcal{DM}$ (and $\mathcal{M}$) analysis.

The abstract constraints are presented in a more readable format than used internally in the analyser : a constraint is written out as a set of modes (MODES), containing pairs of the form (variable,mode), and a minimal set of possible dependencies (PDEPS), i.e. a set of sets of variables where singletons have a special meaning (they indicate the variables having mode a; so there is some overlapping with the MODES component). The dependency information is useful to perform e.g. constraint reordering. Internally, the $\mathcal{DM}$ analyser also keeps track of definite dependencies between variables in order to perform accurate definiteness propagation. As this information is only used for this purpose and not on itself as basis for program optimisation, it is not shown here.

A.1 sumlist

The *sumlist* program specifies the relation between a list of numbers and the sum of its elements. It involves both unification constraints and linear numerical constraints. The annotated normalised program is given below.

```
% Query: sumlist(A,B)
% Entry: MODES: {(A,d),(B,f)}                    PDEPS: ∅
% Exit:  MODES: {(A,d),(B,d)}                    PDEPS: {∅}

% Call:  sumlist(A,B)
% Entry: MODES: {(A,d),(B,f)}                    PDEPS: ∅
% Exit:  MODES: {(A,d),(B,d)}                    PDEPS: {∅}
sumlist(L,S) :-
        AC1  L=[],
        AC2  S=0.   AC3
sumlist(L,S) :-
        AC4  L=[H|T],
        AC5  S=H+S1,
        AC6  sumlist(T,S1).   AC7
```

AC_1	MODES	{(L,d),(S,f)}
	PDEPS	∅
AC_2	MODES	{(L,d),(S,f)}
	PDEPS	{∅}
AC_3	MODES	{(L,d),(S,d)}
	PDEPS	∅

AC_4	MODES	{(H,f),(L,d),(S,f),(S1,f),(T,f)}
	PDEPS	∅
AC_5	MODES	{(H,d),(L,d),(S,f),(S1,f),(T,d)}
	PDEPS	{∅}
AC_6	MODES	{(H,d),(L,d),(S,f),(S1,f),(T,d)}
	PDEPS	{{S,S1}}
AC_7	MODES	{(H,d),(L,d),(S,d),(S1,d),(T,d)}
	PDEPS	{∅}

Note that the projection of the call constraint AC_6 of the recursive call onto the call variables T and $S1$ is identical (up to renaming) to the query pattern. So only one predicate version is obtained.
Possible failure is derived at a number of program points. In AC_2, the definite variable L is unified with the empty list; failure may occur if L had a different value in the query. The same holds for AC_5. In AC_7, the recursive call to sumlist has constrained T to be a list; the analysis then derives possible incompatibility with the value T had before the call.
The derived information allows several program optimisations. In the first clause, AC_1 indicates that L is definite, so $L = 0$ is a simple test. Before executing $S = 0$, S is free (AC_2). If it is also known that S is untyped at that point (S has mode f_u), then $S = 0$ can be transformed into an assignment to S. In the second clause, AC_4 states that L is definite. Moreover, H and T are free and untyped (as it is their first occurrence in the clause). So $L = [H \mid T]$ is a selection operation, for which the compiler can generate specialised code. The numerical constraint $S = H + S1$ can be moved after the recursive call. This is shown by applying Proposition 11.4.3. This proposition is formulated in terms of the freeness abstraction on itself (not involving definiteness information); however, the $\mathcal{M}$ information can easily be obtained from the $\mathcal{DM}$ information through the *extend*^{m} operation (Definition 8.2.3). The superscript e is used to indicate the abstract constraints obtained after extension. The following conditions for constraint reordering are fulfilled :

1. $AC_5^e = \{\emptyset, \{H\}, \{L\}, \{T\}\}$, so $AC_5^e \notin \{\bot, Afail\}$ (i.e. point AC_5^e can be reached);
2. $AC_6^e = \{\{H\}, \{L\}, \{T\}, \{S, S1\}\}$, so $AC_6^e \neq Afail$ and $\emptyset \notin AC_6^e$ (no definite/possible failure occurs by joining $S = H + S1$ to AC_5^e);
3. $AC_7^e = \{\emptyset, \{H\}, \{L\}, \{T\}, \{S\}, \{S1\}\} \neq \bot$ and $\exists^{\mathcal{M}}_{\{T,S1\}} no\_fail(AC_h) = \emptyset$, where
   $\alpha^{\mathcal{M}}(S = H + S1) = \{\{S, H, S1\}\}$ and
   $AC_h = \alpha^{\mathcal{M}}(S = H + S1) \cup (no\_fail(AC_5^e) \oplus^{\mathcal{M}} \alpha^{\mathcal{M}}(S = H + S1))$
   $= \{\{H, S, S1\}, \{L, S, S1\}, \{S, S1\}, \{S, S1, T\}\}$
   ($S = H + S1$ cannot influence the recursive call).

Also note that $\exists^{\mathcal{M}}_{\{T,S1\}} no\_fail(AC_5^e) = \exists^{\mathcal{M}}_{\{T,S1\}} no\_fail(AC_6^e) = \{T\}$, so the reordering will not influence the annotations within the recursive *sumlist* call. After moving the constraint $S = H + S1$ behind the call, the abstract call information of $S = H + S1$ becomes : `MODES:` { (H,d),(L,d),(S,f),(S1,d),(T,d) }, `PDEPS:` {∅}. As H and $S1$ are definite and S is free, if S is known to be untyped then $S = H + S1$ can be transformed into the assignment $S := H + S1$.
Since the first argument L is definite, the constraints involving L ($L = []$ and $L = [H \mid T]$) are mutually exclusive and the predicate is deterministic.
The resulting program code that could be generated by an optimising compiler has been shown in Section 11.4.9.

The analyser output for the non-normalised *sumlist* program is quite similar. The call-head constraint solving is more hidden, although using the call pattern of the predicate allows to derive the same optimisations as mentioned above.

```
% Query: sumlist(A,B)
% Entry: MODES: {(A,d),(B,f)}          PDEPS: ∅
% Exit:  MODES: {(A,d),(B,d)}          PDEPS: {∅}
```

```
% Call:  sumlist(A,B)
% Entry: MODES: {(A,d),(B,f)}                    PDEPS: ∅
% Exit:  MODES: {(A,d),(B,d)}                    PDEPS: {∅}
sumlist([],0).
sumlist([H|T],S) :-
        AC1   S=H+S1,
        AC2   sumlist(T,S1).   AC3
```

AC_1	MODES	{(H,d),(S,f),(S1,f),(T,d)}
	PDEPS	{∅}
AC_2	MODES	{(H,d),(S,f),(S1,f),(T,d)}
	PDEPS	{{S,S1}}
AC_3	MODES	{(H,d),(S,d),(S1,d),(T,d)}
	PDEPS	{∅}

A.2 fib

The *fib* program defines the relation $fib(N, F)$ where F is the N^{th} Fibonacci number. This program contains only linear numerical constraints. Only the normalised version of the program is presented; the output for the non-normalised version is similar.

```
% Query: fib(A,B)
% Entry: MODES: {(A,d),(B,f)}                    PDEPS: ∅
% Exit:  MODES: {(A,d),(B,d)}                    PDEPS: {∅}
```

```
% Call:  fib(A,B)
% Entry: MODES: {(A,d),(B,f)}                    PDEPS: ∅
% Exit:  MODES: {(A,d),(B,d)}                    PDEPS: {∅}
fib(N,F) :-
        AC1   N=0,
        AC2   F=1.   AC3
fib(N,F) :-
        AC4   N=1,
        AC5   F=1.   AC6
fib(N,F) :-
        AC7   N>1,
        AC8   N1=N-1,
        AC9   N2=N-2,
        AC10  F=F1+F2,
        AC11  fib(N1,F1),
        AC12  fib(N2,F2).   AC13
```

AC_1	MODES	{(F,f),(N,d)}
	PDEPS	$\emptyset$
AC_2	MODES	{(F,f),(N,d)}
	PDEPS	{$\emptyset$}
AC_3	MODES	{(F,d),(N,d)}
	PDEPS	$\emptyset$
AC_4	MODES	{(F,f),(N,d)}
	PDEPS	$\emptyset$
AC_5	MODES	{(F,f),(N,d)}
	PDEPS	{$\emptyset$}
AC_6	MODES	{(F,d),(N,d)}
	PDEPS	$\emptyset$
AC_7	MODES	{(F,f),(F1,f),(F2,f),(N,d),(N1,f),(N2,f)}
	PDEPS	$\emptyset$
AC_8	MODES	{(F,f),(F1,f),(F2,f),(N,d),(N1,f),(N2,f)}
	PDEPS	{$\emptyset$}
AC_9	MODES	{(F,f),(F1,f),(F2,f),(N,d),(N1,d),(N2,f)}
	PDEPS	$\emptyset$
AC_{10}	MODES	{(F,f),(F1,f),(F2,f),(N,d),(N1,d),(N2,d)}
	PDEPS	$\emptyset$
AC_{11}	MODES	{(F,f),(F1,f),(F2,f),(N,d),(N1,d),(N2,d)}
	PDEPS	{{F,F1,F2}}
AC_{12}	MODES	{(F,f),(F1,d),(F2,f),(N,d),(N1,d),(N2,d)}
	PDEPS	{$\emptyset$,{F,F2}}
AC_{13}	MODES	{(F,d),(F1,d),(F2,d),(N,d),(N1,d),(N2,d)}
	PDEPS	{$\emptyset$}

The projection of AC_{11} and AC_{12} onto respectively the variables $\{N1, F1\}$ and $\{N2, F2\}$ yields the same entry pattern for the recursive calls as the original query pattern. Hence only one predicate version is derived.

Possible failure occurs when additional constraints are put upon the definite variables N_1 or N_2, as is the case in AC_2, AC_5, AC_8, AC_{12} and AC_{13}.

Several program optimisations are possible based on the analyser output. The constraints $N = 0$, $N = 1$ and $N > 1$ can be transformed into simple tests, as N is known to be definite by the time these constraints are set up. Especially in the case of the inequality, specialisation is worthwhile. It avoids the invocation of the simplex algorithm. The constraints $N1 = N - 1$ and $N2 = N - 2$ can be transformed into assignments. Also, $F = 1$ in the first and in the second clause can be turned into an assignment to the free variable F, at least if F is known to be untyped before executing the constraint. The most important optimisation consists of moving the constraint $F = F1 + F2$ over the two recursive *fib* calls to the end of the clause. The safety of this reordering is based on Proposition 11.4.3, using the extended abstract constraint information (i.e. the $\mathcal{M}$ output obtained from the $\mathcal{DM}$ output). First, consider the reordering of $F = F1 + F2, fib(N1, F1)$ to $fib(N1, F1), F = F1 + F2$; the following criteria are satisfied :

1. $AC^e_{10} = \{\{N\},\{N1\},\{N2\}\} \not\in \{\bot, Afail\}$;

2. $AC^e_{11} = \{\{N\},\{N1\},\{N2\},\{F,F1,F2\}\}$, so $AC^e_{11} \neq Afail$ and $\emptyset \not\in AC^e_{11}$;

3. $AC^e_{12} \neq \bot$ and $\exists^{\mathcal{M}}_{\{N1,F1\}} no\_fail(AC_h) = \emptyset$, where
   $\alpha^{\mathcal{M}}(F = F1 + F2) = \{\{F,F1,F2\}\}$ and
   $AC_h = \alpha^{\mathcal{M}}(F = F1 + F2) \cup (no\_fail(AC^e_{10}) \oplus^{\mathcal{M}} \alpha^{\mathcal{M}}(F = F1 + F2))$
   $= \{\{F,F1,F2\},\{F,F1,F2,N\},\{F,F1,F2,N1\},\{F,F1,F2,N2\}\}$.

Note that $\exists^{\mathcal{M}}_{\{N1,F1\}} no\_fail(AC^e_{10}) = \exists^{\mathcal{M}}_{\{N1,F1\}} no\_fail(AC^e_{11}) = \{N1\}$, so the reordering cannot influence the annotations within the recursive call $fib(N1,F1)$. After the first reordering, we obtain

$$\ldots, AC^e_{10} \quad \mathbf{fib(N1,F1)}, \quad AC^e_{11b} \quad \mathbf{F = F1 + F2}, \quad AC^e_{12b} \quad \mathbf{fib(N2,F2)} \quad AC^e_{13}$$

with $AC^e_{11b} = \{\emptyset,\{F1\},\{N\},\{N1\},\{N2\}\}$, $AC^e_{12b} = \{\{F,F2\},\{F1\},\{N\},\{N1\},\{N2\}\}$. Again, $F = F1 + F2$ can be moved over $fib(N2,F2)$ applying the same proposition but now based on AC^e_{11b}, AC^e_{12b} and AC^e_{13}. When $F = F1 + F2$ is at the end of the clause, its call constraint yields the following mode information : $\{(F,f),(F1,d),(F2,d),(N,d),(N1,d),(N2,d)\}$. So both $F1$ and $F2$ have become definite and F is still free. If F is also known to be untyped at that point, then the constraint can be turned into an assignment to F.
Finally, the definiteness of the first argument N implies that the constraints $N = 0$, $N = 1$ and $N > 1$ are mutually exclusive, which allows to generate deterministic code for fib.

A.3 mortgage

The well-known *mortgage* program involves both linear and non-linear numerical constraints. As the analysis gives rise to different predicate versions, we also present the original (normalised) program. In the analysis, the more precise abstraction of non-linear constraints is used (cf. Section 9.3).

Original program :

```
mortgage(P,T,I,B,MP) :-
        T=1,
        B=P+P*I-MP.
mortgage(P,T,I,B,MP) :-
        T>1,
        T1=T-1,
        P1=P*(1+I)-MP,
        mortgage(P1,T1,I,B,MP).
```

Analyser output :

```
% Query: mortgage(P,T,I,B,MP)
% Entry: MODES: {(P,f),(T,d),(I,d),(B,d),(MP,d)} PDEPS: ∅
% Exit:  MODES: {(P,a),(T,d),(I,d),(B,d),(MP,d)} PDEPS: {∅,{P}}


% Call:  mortgage(A,B,C,D,E)
% Entry: MODES: {(A,f),(B,d),(C,d),(D,d),(E,d)}  PDEPS: ∅
% Exit:  MODES: {(A,a),(B,d),(C,d),(D,d),(E,d)}  PDEPS: {∅,{A}}
mortgage(P,T,I,B,MP) :-
        AC1   T=1,
        AC2   B=P+P*I-MP.   AC3
mortgage(P,T,I,B,MP) :-
        AC4   T>1,
        AC5   T1=T-1,
        AC6   P1=P*(1+I)-MP,
        AC7   mortgage_2(P1,T1,I,B,MP).   AC8
```

```
% Call:  mortgage_2(A,B,C,D,E)
% Entry: MODES: {(A,a),(B,d),(C,d),(D,d),(E,d)}  PDEPS: {{A}}
% Exit:  MODES: {(A,a),(B,d),(C,d),(D,d),(E,d)}  PDEPS: {∅,{A}}
mortgage_2(P,T,I,B,MP) :-
        AC9   T=1,
        AC10  B=P+P*I-MP.   AC11
mortgage_2(P,T,I,B,MP) :-
        AC12  T>1,
        AC13  T1=T-1,
        AC14  P1=P*(1+I)-MP,
        AC15  mortgage_2(P1,T1,I,B,MP).   AC16
```

The recursive call $mortgage(P1, T1, I, B, MP)$ in the second clause has an entry pattern that differs from the original query pattern : $P1$ has mode a instead of f (compare AC_7 (projected onto the call variables) with the query pattern). This gives rise to another predicate version *mortgage_2*. The recursive call in *mortgage_2* has the same entry pattern (the projected AC_{15} is identical to the projected AC_7), so *mortgage_2* does not yield further predicate versions.
As for the *sumlist* and *fib* example, the possible failure information is due to further constraining of definite variables.

mortgage/5		
AC_1	MODES	{(B,d),(I,d),(MP,d),(P,f),(T,d)}
	PDEPS	∅
AC_2	MODES	{(B,d),(I,d),(MP,d),(P,f),(T,d)}
	PDEPS	{∅}
AC_3	MODES	{(B,d),(I,d),(MP,d),(P,a),(T,d)}
	PDEPS	{∅,{P}}
AC_4	MODES	{(B,d),(I,d),(MP,d),(P,f),(P1,f),(T,d),(T1,f)}
	PDEPS	∅
AC_5	MODES	{(B,d),(I,d),(MP,d),(P,f),(P1,f),(T,d),(T1,f)}
	PDEPS	{∅}
AC_6	MODES	{(B,d),(I,d),(MP,d),(P,f),(P1,f),(T,d),(T1,d)}
	PDEPS	∅
AC_7	MODES	{(B,d),(I,d),(MP,d),(P,f),(P1,a),(T,d),(T1,d)}
	PDEPS	{{P,P1},{P1}}
AC_8	MODES	{(B,d),(I,d),(MP,d),(P,a),(P1,a),(T,d),(T1,d)}
	PDEPS	{∅,{P},{P1}}

mortgage_2/5		
AC_9	MODES	{(B,d),(I,d),(MP,d),(P,a),(T,d)}
	PDEPS	{{P}}
AC_{10}	MODES	{(B,d),(I,d),(MP,d),(P,a),(T,d)}
	PDEPS	{∅,{P}}
AC_{11}	MODES	{(B,d),(I,d),(MP,d),(P,a),(T,d)}
	PDEPS	{∅,{P}}
AC_{12}	MODES	{(B,d),(I,d),(MP,d),(P,a),(P1,f),(T,d),(T1,f)}
	PDEPS	{{P}}
AC_{13}	MODES	{(B,d),(I,d),(MP,d),(P,a),(P1,f),(T,d),(T1,f)}
	PDEPS	{∅,{P}}
AC_{14}	MODES	{(B,d),(I,d),(MP,d),(P,a),(P1,f),(T,d),(T1,d)}
	PDEPS	{{P}}
AC_{15}	MODES	{(B,d),(I,d),(MP,d),(P,a),(P1,a),(T,d),(T1,d)}
	PDEPS	{{P},{P1}}
AC_{16}	MODES	{(B,d),(I,d),(MP,d),(P,a),(P1,a),(T,d),(T1,d)}
	PDEPS	{∅,{P},{P1}}

The minimal freeness abstraction of the non-linear constraint $P1 = P * (1 + I) - MP$ in the second clause is $\{\{P1, P, MP\}, \{P1, I, MP\}\}$, at least if the more precise abstraction of non-linear constraints is used (the imprecise abstraction is $\{\{P1\}, \{P\}, \{I\}, \{MP\}\}$). The first time (for the *mortgage* version), the call pattern of this constraint is given by AC_6. It indicates that I and MP are definite and P and $P1$ are free. Joining AC_6 with the abstraction of the non-linear constraint yields AC_7, which indicates that $P1$ is possibly non-free and there is a dependency between P and $P1$. This is the most precise abstraction that can be obtained if no information on the values of I and MP is available. Indeed, in the special case that I has the value -1, the constraint reduces to $P1 = MP$ and hence $P1$ obtains the value of MP. If I is different from -1 (as can be assumed in a concrete call to *mortgage*), then $P1$ remains free. In *mortage_2*, a similar situation occurs.

Before setting up the non-linear constraint, I and MP are definite, $P1$ is free and now P is already possibly non-free. Again, if I has the value -1, the non-linear constraint reduces to $P1 = MP$, causing $P1$ to become non-free.
An analogous reasoning holds for the linear constraint $B = P + P * I - MP$ in the first clause. It is known that B, I and MP are definite by the time this constraint is reached (cf. AC_2 and AC_{10}). If $I = -1$, the constraint reduces to the test $B = MP$; otherwise, it reduces to $B = P * (1 + n) - MP$ with n a number different from -1. In the first case, P remains free, whereas in the second case P becomes definite. So the analysis can only derive that P becomes possibly non-free (although in a "normal" call to mortgage I may be assumed different from -1, so then P becomes definite).

We sum up the optimisations that can be performed using the derived information. First of all, the constraints $T = 1$ and $T > 1$ in *mortgage* as well as in *mortgage_2* can be translated into simple tests as T is definite by the time these constraints are encountered; also, $T1 = T - 1$ can be transformed into an assignment to the free and untyped variable $T1$ ($T1$ is known to be untyped as it is the first time it occurs within the clause).
Secondly, since the second argument T is always definite, the constraints $T = 1$ and $T > 1$ are mutually exclusive; so deterministic code can be generated.
Thirdly, the non-linear constraints are all linear by the time they are set up as I is known to be definite at those points (AC_2, AC_6, AC_{10} and AC_{14}).
Checking the conditions of Proposition 11.4.3 for putting the constraint $P1 = P * (1 + I) - MP$ after the recursive call shows that reordering is not possible in this case. Indeed, $AC_h = \alpha^{\mathcal{M}}(P1 = P * (1 + I) - MP) \cup (no\_fail(AC_6^e) \oplus^{\mathcal{M}} \alpha^{\mathcal{M}}(P1 = P * (1 + I) - MP)) = \{\{B, P1\}, \{I, P1\}, \{MP, P1\}, \{P, P1\}, \{P1\}, \{P1, T\}, \{P1, T1\}\}$, and therefore $\exists^{\mathcal{M}}_{\{P1,T1,I,B,MP\}} no\_fail(AC_h) = \{\{B, P1\}, \{I, P1\}, \{MP, P1\}, \{P1\}, \{P1, T1\}\} \neq \emptyset$. Even using the stronger Proposition 11.4.2 does not allow to perform constraint reordering. Note that, if I were known to be different from -1, as is normally the case in a call to *mortgage*, joining the constraint with its call pattern would yield a dependency between P and $P1$ and leave both variables free. Since only $P1$ is involved in the recursive call, the non-linear constraint also involving the free variable P would not be able to interfere with the recursive computation and hence it could safely be put after the recursive call. At that point it could be transformed into an assignment to P ($P1$ would have become definite by then).

A.4 vecmat

The *vecmat* program performs vector and matrix operations. A vector is represented as a PrologIII tuple, a matrix as a tuple of tuples. The query pattern considered is $main(d, d)$ which gives rise to the call patterns $matvecmul(d, d, f)$, $vecadd(f, d, d)$ and $matmul(d, d, f)$ (*matvecmul* specifies the multiplication of a matrix with a vector, *vecadd* the addition of two vectors, and *matmul* defines matrix multiplication). The program contains tuple constraints and linear and non-linear numerical constraints. We present the more compact non-normalised version of the program; the analysis of the normalised version gives similar results. As for the *mortgage* program, the more precise abstraction of non-linear constraints is considered (cf. Section 9.3). This results in abstract information that is 100% precise. The abstraction allows to reorder the non-linear constraints as is shown further, whereas this could not be inferred using the imprecise abstraction (which

derives mode a for variables involved in a non-linear constraint).

```
% Query: main(M1,V1)
% Entry: MODES: {(M1,d),(V1,d)}                PDEPS: ∅
% Exit:  MODES: {(M1,d),(V1,d)}                PDEPS: {∅}


% Call:  vecadd(A,B,C)
% Entry: MODES: {(A,f),(B,d),(C,d)}            PDEPS: ∅
% Exit:  MODES: {(A,d),(B,d),(C,d)}            PDEPS: {∅}
vecadd(⟨⟩, ⟨⟩, ⟨⟩).   AC1
vecadd(⟨H1⟩.T1, ⟨H2⟩.T2, ⟨H1+H2⟩.T3) :-
        AC2  vecadd(T1,T2,T3).   AC3

% Call:  matvecmul(A,B,C)
% Entry: MODES: {(A,d),(B,d),(C,f)}            PDEPS: ∅
% Exit:  MODES: {(A,d),(B,d),(C,d)}            PDEPS: {∅}
matvecmul(⟨⟩, V, ⟨⟩).   AC4
matvecmul(⟨Z⟩.R, V, ⟨H⟩.T) :-
        AC5  skalar(Z,V,H),
        AC6  matvecmul(R,V,T).   AC7

% Call:  matmul(A,A,B)
% Entry: MODES: {(A,d),(B,f)}                  PDEPS: ∅
% Exit:  MODES: {(A,d),(B,d)}                  PDEPS: {∅}
matmul(⟨⟩, A, ⟨⟩).   AC8
matmul(⟨Z1⟩.Z2, M, ⟨Z3⟩.Z4) :-
        AC9  vecmatmul(Z1,M,Z3),
        AC10 matmul_2(Z2,M,Z4).   AC11

% Call:  matmul_2(A,B,C)
% Entry: MODES: {(A,d),(B,d),(C,f)}            PDEPS: ∅
% Exit:  MODES: {(A,d),(B,d),(C,d)}            PDEPS: {∅}
matmul_2(⟨⟩, A, ⟨⟩).   AC12
matmul_2(⟨Z1⟩.Z2, M, ⟨Z3⟩.Z4) :-
        AC13 vecmatmul(Z1,M,Z3),
        AC14 matmul_2(Z2,M,Z4).   AC15

% Call:  vecmatmul(A,B,C)
% Entry: MODES: {(A,d),(B,d),(C,f)}            PDEPS: ∅
% Exit:  MODES: {(A,d),(B,d),(C,d)}            PDEPS: {∅}
vecmatmul(V, M, ⟨Z1⟩.Z2) :-
        AC16 el(V,M,Z1,M1),
        AC17 vecmatmul(V,M1,Z2).   AC18
vecmatmul(X, R, ⟨⟩) :-
        AC19 empty(R).   AC20
```

```
% Call:  empty(A)
% Entry: MODES: {(A,d)}                          PDEPS: ∅
% Exit:  MODES: {(A,d)}                          PDEPS: {∅}
empty((⟨⟩).R) :-
        AC21   empty(R).   AC22
empty(⟨⟩).   AC23

% Call:  el(A,B,C,D)
% Entry: MODES: {(A,d),(B,d),(C,f),(D,f)}        PDEPS: ∅
% Exit:  MODES: {(A,d),(B,d),(C,d),(D,d)}        PDEPS: {∅}
el(⟨⟩, ⟨⟩, 0, ⟨⟩).   AC24
el((V1).V2, ⟨(Z1).Z2⟩.Z3, P, ⟨Z2⟩.Z4) :-
        AC25   P=P1+V1*Z1,
        AC26   el(V2,Z3,P1,Z4).   AC27

% Call:  skalar(A,B,C)
% Entry: MODES: {(A,d),(B,d),(C,f)}              PDEPS: ∅
% Exit:  MODES: {(A,d),(B,d),(C,d)}              PDEPS: {∅}
skalar(⟨⟩, ⟨⟩, 0).   AC28
skalar(⟨H1⟩.T1, ⟨H2⟩.T2, P) :-
        AC29   P=T3+H1*H2,
        AC30   skalar(T1,T2,T3).   AC31

% Call:  main(A,B)
% Entry: MODES: {(A,d),(B,d)}                    PDEPS: ∅
% Exit:  MODES: {(A,d),(B,d)}                    PDEPS: ∅
main(M1,V1) :-
        AC32   matvecmul(M1,V1,V2),
        AC33   vecadd(V3,V1,V2),
        AC34   matmul(M1,M1,M2).   AC35
```

The analysis gives rise to two versions of the *matmul*/3 predicate. The specialisation occurs because of the specialisation criterion applied in the abstract interpretation procedure : a predicate call is further analysed if it is syntactically different (up to renaming) from the previously encountered calls to that predicate or if its abstract entry constraint is different. In the *matmul* case, the first call $matmul(A, A, B)$ syntactically differs from the second call $matmul\_2(A, B, C)$. However, the abstract information in the clauses of the two versions is the same (AC_8, AC_9, AC_{10} and AC_{11} are identical to respectively AC_{12}, AC_{13}, AC_{14} and AC_{15}). Hence, the same code will be generated for the two versions. A more sophisticated specialisation criterion could be used to avoid specialisation in such a case.

vecadd/3		
AC_1	MODES	$\emptyset$
	PDEPS	{$\emptyset$}
AC_2	MODES	{(H1,d),(H2,d),(T1,f),(T2,d),(T3,d)}
	PDEPS	{$\emptyset$}
AC_3	MODES	{(H1,d),(H2,d),(T1,d),(T2,d),(T3,d)}
	PDEPS	{$\emptyset$}

matvecmul/3		
AC_4	MODES	{(V,d)}
	PDEPS	{$\emptyset$}
AC_5	MODES	{(H,f),(R,d),(T,f),(V,d),(Z,d)}
	PDEPS	{$\emptyset$}
AC_6	MODES	{(H,d),(R,d),(T,f),(V,d),(Z,d)}
	PDEPS	{$\emptyset$}
AC_7	MODES	{(H,d),(R,d),(T,d),(V,d),(Z,d)}
	PDEPS	{$\emptyset$}

matmul/3		
AC_8	MODES	{(A,d)}
	PDEPS	{$\emptyset$}
AC_9	MODES	{(M,d),(Z1,d),(Z2,d),(Z3,f),(Z4,f)}
	PDEPS	{$\emptyset$}
AC_{10}	MODES	{(M,d),(Z1,d),(Z2,d),(Z3,d),(Z4,f)}
	PDEPS	{$\emptyset$}
AC_{11}	MODES	{(M,d),(Z1,d),(Z2,d),(Z3,d),(Z4,d)}
	PDEPS	{$\emptyset$}

matmul_2/3		
AC_{12}	MODES	{(A,d)}
	PDEPS	{$\emptyset$}
AC_{13}	MODES	{(M,d),(Z1,d),(Z2,d),(Z3,f),(Z4,f)}
	PDEPS	{$\emptyset$}
AC_{14}	MODES	{(M,d),(Z1,d),(Z2,d),(Z3,d),(Z4,f)}
	PDEPS	{$\emptyset$}
AC_{15}	MODES	{(M,d),(Z1,d),(Z2,d),(Z3,d),(Z4,d)}
	PDEPS	{$\emptyset$}

vecmatmul/3		
AC_{16}	MODES	{(M,d),(M1,f),(V,d),(Z1,f),(Z2,f)}
	PDEPS	∅
AC_{17}	MODES	{(M,d),(M1,d),(V,d),(Z1,d),(Z2,f)}
	PDEPS	{∅}
AC_{18}	MODES	{(M,d),(M1,d),(V,d),(Z1,d),(Z2,d)}
	PDEPS	{∅}
AC_{19}	MODES	{(R,d),(X,d)}
	PDEPS	∅
AC_{20}	MODES	{(R,d),(X,d)}
	PDEPS	{∅}

empty/1		
AC_{21}	MODES	{(R,d)}
	PDEPS	{∅}
AC_{22}	MODES	{(R,d)}
	PDEPS	{∅}
AC_{23}	MODES	∅
	PDEPS	{∅}

el/4		
AC_{24}	MODES	∅
	PDEPS	{∅}
AC_{25}	MODES	{(P,f),(P1,f),(V1,d),(V2,d),(Z1,d),(Z2,d),(Z3,d),(Z4,f)}
	PDEPS	{∅}
AC_{26}	MODES	{(P,f),(P1,f),(V1,d),(V2,d),(Z1,d),(Z2,d),(Z3,d),(Z4,f)}
	PDEPS	{{P,P1}}
AC_{27}	MODES	{(P,d),(P1,d),(V1,d),(V2,d),(Z1,d),(Z2,d),(Z3,d),(Z4,d)}
	PDEPS	{∅}

skalar/3		
AC_{28}	MODES	∅
	PDEPS	{∅}
AC_{29}	MODES	{(H1,d),(H2,d),(P,f),(T1,d),(T2,d),(T3,f)}
	PDEPS	{∅}
AC_{30}	MODES	{(H1,d),(H2,d),(P,f),(T1,d),(T2,d),(T3,f)}
	PDEPS	{{P,T3}}
AC_{31}	MODES	{(H1,d),(H2,d),(P,d),(T1,d),(T2,d),(T3,d)}
	PDEPS	{∅}

main/2		
AC_{32}	MODES	{(M1,d),(M2,f),(V1,d),(V2,f),(V3,f)}
	PDEPS	$\emptyset$
AC_{33}	MODES	{(M1,d),(M2,f),(V1,d),(V2,d),(V3,f)}
	PDEPS	{$\emptyset$}
AC_{34}	MODES	{(M1,d),(M2,f),(V1,d),(V2,d),(V3,d)}
	PDEPS	{$\emptyset$}
AC_{35}	MODES	{(M1,d),(M2,d),(V1,d),(V2,d),(V3,d)}
	PDEPS	{$\emptyset$}

We briefly outline the possible program optimisations based on the derived information. One of the optimisations is constraint specialisation : part of the tuple constraints are tests or selections, and the other part are constructions. For example, in the second clause of *vecadd*, the hidden constraints involved in call-head matching are $A = (H1).T1$, $B = (H2).T2$, $C = (H3).T3$ and $H3 = H1 + H2$, where the A, B and C indicate the call arguments. The second and third argument (B and C) are definite when *vecadd* is called, so this results in testing whether these arguments are non-empty tuples and selecting tuple components. Also, the numerical constraint $H3 = H1 + H2$ can be transformed into an assignment of the value $H3 - H2$ to $H1$, as $H2$ and $H3$ are known to be definite and $H1$ is free and untyped. The first constraint $A = (H1).T1$ involves a tuple creation. It is best to reorder the primitive constraints such that selections precede tuple creation.
Other optimisations consist of making use of determinacy of predicates (similar as in the previous examples) and of exploiting the knowledge that the non-linear constraints in *el*/4 and *skalar*/3 are linear by the time they are reached (based on the definiteness of the variables $V1$ and $Z1$, respectively $H1$ and $H2$).
Finally, the non-linear constraints can be moved over the recursive calls to the end of the clauses in which they appear. As in the previous examples, this is based on applying Proposition 11.4.3. For example, for $P = T3 + H1 * H2$ in *skalar*/3 we have :

1. $AC^{\epsilon}_{29} = \{\{H1\},\{H2\},\{T1\},\{T2\}\} \not\in \{\bot, Afail\}$;
2. $AC^{\epsilon}_{30} = \{\{H1\},\{H2\},\{P,T3\},\{T1\},\{T2\}\}$, so $AC^{\epsilon}_{30} \neq Afail$ and $\emptyset \not\in AC^{\epsilon}_{30}$
3. $AC^{\epsilon}_{31} = \{\{H1\},\{H2\},\{P\},\{T1\},\{T2\},\{T3\}\} \neq \bot$ and $\exists^{\mathcal{M}}_{\{T1,T2,T3\}} no\_fail(AC_h) = \emptyset$ where
   $AC_h = \alpha^{\mathcal{M}}(P = T3 + H1 * H2) \cup (no\_fail(AC^{\epsilon}_{29}) \oplus^{\mathcal{M}} \alpha^{\mathcal{M}}(P = T3 + H1 * H2))$
   $= \{\{H1,P,T3\},\{H2,P,T3\},\{P,T1,T3\},\{P,T2,T3\},\{P,T3\}\}$.

After the reordering, the non-linear constraint can be transformed into an assignment to P (at least if P is known to be untyped), as $T3$ will have become definite by then and $H1$ and $H2$ were already definite. A similar reasoning holds for $P = P1 + V1 * Z1$ in *el*/4.

A.5 runkut

The *runkut* program is a program for first-order differential equation solving using the runge-kutta method. It involves unification constraints and linear and non-linear numerical constraints.

```
    % Query: solve(T1,H,Xend)
    % Entry: MODES: {(T1,d),(H,d),(Xend,f)}        PDEPS: ∅
    % Exit:  MODES: {(T1,d),(H,d),(Xend,a)}        PDEPS: {∅,{Xend}}

% Call:  solve(A,B,C)
% Entry: MODES: {(A,d),(B,d),(C,f)}              PDEPS: ∅
% Exit:  MODES: {(A,d),(B,d),(C,a)}              PDEPS: {∅,{C}}
solve([X0,Y0],H,Xend) :-
        AC1  loop([X0,Y0],H,[X1,Y1],Xend).  AC2

% Call:  loop([A,B],C,[D,E],F)
% Entry: MODES: {(A,d),(B,d),(C,d),(D,f),(E,f),(F,f)}    PDEPS: ∅
% Exit:  MODES: {(A,d),(B,d),(C,d),(D,d),(E,a),(F,a)}    PDEPS: {{E},{F}}
loop([Xi,Yi],H,[Xi1,Yi1],Xend) :-
        AC3  Xi1=Xi+H,
        AC4  Xi1>Xend.  AC5
loop([Xi,Yi],H,[Xi1,Yi1],Xend) :-
        AC6  rk([Xi,Yi],H,[Xi1,Yi1]),
        AC7  loop([Xi1,Yi1],H,[Xi2,Yi2],Xend).  AC8

% Call:  rk([A,B],C,[D,E])
% Entry: MODES: {(A,d),(B,d),(C,d),(D,f),(E,f)}  PDEPS: ∅
% Exit:  MODES: {(A,d),(B,d),(C,d),(D,d),(E,d)}  PDEPS: ∅
rk([Xi,Yi],H,[Xi1,Yi1]) :-
        AC9  Xi1=Xi+H,
        AC10 Yi1=Yi+(V1+2*V2+2*V3+V4)*H/6,
        AC11 ode_3(Xi,Yi,V1),
        AC12 ode_2(Xi+H/2,Yi+H*V1/2,V2),
        AC13 ode_2(Xi+H/2,Yi+H*V2/2,V3),
        AC14 ode(Xi1,Yi+H*V3,V4).  AC15

% Call:  ode(A,B+C*D,E)
% Entry: MODES: {(A,d),(B,d),(C,d),(D,d),(E,f)}  PDEPS: ∅
% Exit:  MODES: {(A,d),(B,d),(C,d),(D,d),(E,d)}  PDEPS: ∅
ode(X,Y,X-Y*Y).  AC16

% Call:  ode_2(A+B/2,C+B*D/2,E)
% Entry: MODES: {(A,d),(B,d),(C,d),(D,d),(E,f)}  PDEPS: ∅
% Exit:  MODES: {(A,d),(B,d),(C,d),(D,d),(E,d)}  PDEPS: ∅
ode_2(X,Y,X-Y*Y).  AC17

% Call:  ode_3(A,B,C)
% Entry: MODES: {(A,d),(B,d),(C,f)}              PDEPS: ∅
% Exit:  MODES: {(A,d),(B,d),(C,d)}              PDEPS: ∅
ode_3(X,Y,X-Y*Y).  AC18
```

There are three syntactically different calls to *ode*, giving rise to three specialised versions *ode*, *ode_2* and *ode_3*. Note however that the abstract constraints within the three versions are identical, so it would be desirable to avoid specialisation.

solve/3		
AC_1	MODES	{(H,d),(X0,d),(X1,f),(Xend,f),(Y0,d),(Y1,f)}
	PDEPS	{∅}
AC_2	MODES	{(H,d),(X0,d),(X1,d),(Xend,a),(Y0,d),(Y1,a)}
	PDEPS	{{Xend},{Y1}}

loop/4		
AC_3	MODES	{(H,d),(Xend,f),(Xi,d),(Xi1,f),(Yi,d),(Yi1,f)}
	PDEPS	∅
AC_4	MODES	{(H,d),(Xend,f),(Xi,d),(Xi1,d),(Yi,d),(Yi1,f)}
	PDEPS	∅
AC_6	MODES	{(H,d),(Xend,a),(Xi,d),(Xi1,d),(Yi,d),(Yi1,f)}
	PDEPS	{{Xend}}
AC_6	MODES	{(H,d),(Xend,f),(Xi,d),(Xi1,f),(Xi2,f),(Yi,d),(Yi1,f),(Yi2,f)}
	PDEPS	∅
AC_7	MODES	{(H,d),(Xend,f),(Xi,d),(Xi1,d),(Xi2,f),(Yi,d),(Yi1,d),(Yi2,f)}
	PDEPS	∅
AC_8	MODES	{(H,d),(Xend,a),(Xi,d),(Xi1,d),(Xi2,d),(Yi,d),(Yi1,d),(Yi2,a)}
	PDEPS	{{Xend},{Yi2}}

rk/3		
AC_9	MODES	{(H,d),(V1,f),(V2,f),(V3,f),(V4,f),(Xi,d),(Xi1,f),(Yi,d),(Yi1,f)}
	PDEPS	∅
AC_{10}	MODES	{(H,d),(V1,f),(V2,f),(V3,f),(V4,f),(Xi,d),(Xi1,d),(Yi,d),(Yi1,f)}
	PDEPS	∅
AC_{11}	MODES	{(H,d),(V1,f),(V2,f),(V3,f),(V4,f),(Xi,d),(Xi1,d),(Yi,d),(Yi1,a)}
	PDEPS	{{V1,Yi1},{V2,Yi1},{V3,Yi1},{V4,Yi1},{Yi1}}
AC_{12}	MODES	{(H,d),(V1,d),(V2,f),(V3,f),(V4,f),(Xi,d),(Xi1,d),(Yi,d),(Yi1,a)}
	PDEPS	{{V2,Yi1},{V3,Yi1},{V4,Yi1},{Yi1}}
AC_{13}	MODES	{(H,d),(V1,d),(V2,d),(V3,f),(V4,f),(Xi,d),(Xi1,d),(Yi,d),(Yi1,a)}
	PDEPS	{{V3,Yi1},{V4,Yi1},{Yi1}}
AC_{14}	MODES	{(H,d),(V1,d),(V2,d),(V3,d),(V4,f),(Xi,d),(Xi1,d),(Yi,d),(Yi1,a)}
	PDEPS	{{V4,Yi1},{Yi1}}
AC_{15}	MODES	{(H,d),(V1,d),(V2,d),(V3,d),(V4,d),(Xi,d),(Xi1,d),(Yi,d),(Yi1,d)}
	PDEPS	∅

ode/3		
AC_{16}	MODES	{(X,d),(Y,d)}
	PDEPS	∅

ode_2/3		
AC_{17}	MODES	{(X,d),(Y,d)}
	PDEPS	∅

ode_3/3		
AC_{18}	MODES	{(X,d),(Y,d)}
	PDEPS	$\emptyset$

The program optimisations are not discussed in detail as they are similar to the ones in the previous examples. They include

1. constraint specialisation : the transformation of general unification constraints to tests and (list) selections, or to assignments and (list) constructions; and the transformation of numerical constraints to tests or assignments (e.g. $Xi1 = Xi + H$ in the first clause of *loop*/4 can be turned into an assignment);

2. exploiting the linearity of non-linear constraints by the time they are reached (cf. the *ode* versions);

3. constraint reordering : despite the imprecise abstraction of the non-linear constraint $Yi1 = Yi+(V1+2*V2+2*V3+2*V4)*H/6$, i.e. $\alpha^{\mathcal{M}}(Yi1 = Yi+(V1+2*V2+2*V3+2*V4)*H/6) = \{\{H, Yi, Yi1\}, \{V1, Yi, Yi1\}, \{V2, Yi, Yi1\}, \{V3, Yi, Yi1\}, \{V4, Yi, Yi1\}\}$, the abstract information nevertheless allows to move the constraint over the *ode* calls towards the end of the *rk*/3 clause, where it can then be specialised into an assignment to $Yi1$ (at least if $Yi1$ is known to be untyped). The reordering is based on the application of Proposition 11.4.3 and is similar to the successive reorderings in the *fib* example.

A.6 num

The *num* program transforms numbers into a sequence of letters and phonemes. It contains PrologII tuple constraints and numerical constraints. We do not present the full program but pick out one of the predicates which is representative for the rest and which allows to show the kind of program optimisations that can be performed.

```
% Query: nombre(N,W,P)
% Entry: MODES: {(N,d),(W,f),(P,f)}          PDEPS: ∅
% Exit:  MODES: {(N,d),(W,d),(P,d)}          PDEPS: {∅}
```

...

```
% Call:  moinsMille(A,B,C)
% Entry: MODES: {(A,d),(B,f),(C,f)}          PDEPS: ∅
% Exit:  MODES: {(A,d),(B,d),(C,d)}          PDEPS: {∅}
moinsMille(N,W,P) :-
        AC1  U>=2,
        AC2  N=U*100,
        AC3  W=W1.⟨'cents '⟩,
        AC4  P=P1.⟨ss,an⟩,
        AC5  uniteCC_2(U,W1,P1).    AC6
```

AC_1	MODES	{(N,d),(P,f),(P1,f),(U,f),(W,f),(W1,f)}
	PDEPS	$\emptyset$
AC_2	MODES	{(N,d),(P,f),(P1,f),(U,a),(W,f),(W1,f)}
	PDEPS	{{U}}
AC_3	MODES	{(N,d),(P,f),(P1,f),(U,d),(W,f),(W1,f)}
	PDEPS	{$\emptyset$}
AC_4	MODES	{(N,d),(P,f),(P1,f),(U,d),(W,a),(W1,f)}
	PDEPS	{{W},{W,W1}}
AC_5	MODES	{(N,d),(P,a),(P1,f),(U,d),(W,a),(W1,f)}
	PDEPS	{{P},{P,P1},{W},{W,W1}}
AC_6	MODES	{(N,d),(P,d),(P1,d),(U,d),(W,d),(W1,d)}
	PDEPS	{$\emptyset$}

The primitive constraints $U >= 2$ and $N = U * 100$ should be reordered as the constraint $N = U * 100$ determines the value of U (since N is definite); after reordering, $U >= 2$ turns into a simple test, which avoids the invocation of the simplex algorithm. The constraints $W = W1 \cdot (\textit{'cents'})$ and $P = P1 \cdot \langle ss, an \rangle$ can safely be moved to the end of the clause by applying Proposition 11.4.3. The situation is similar to the reordering in the *sumlist* example. The constraints can then be turned into assignments to the free variables W and P, as $W1$ and $P1$ will have become definite due to *uniteCC 2*. Moreover, in the PrologIII system, a tuple constraint involving concatenation is delayed until the size of the leftmost operand in the concatenation is known. By moving the tuple constraints to the end of the clause, the leftmost operands $W1$ and $P1$ in the concatenations will have become definite; so the constraints do no longer have to be delayed, resulting in more efficient code.

A.7 rectangle

The *rectangle* program specifies how to fill a rectangle of dimension $1 \times b$ with n squares. It involves PrologIII tuple constraints and linear numerical constraints. We only present a part of the program to illustrate how the analysis looses precision due to the absence of structure information.

```
% Query: fillRectangle(A,C)
% Entry: MODES: {(A,f),(C,a)}                    PDEPS: {{C}}
% Exit:  MODES: {(A,a),(C,a)}                    PDEPS: {∅,{A},{C}}


% Call:  fillRectangle(A,B)
% Entry: MODES: {(A,f),(B,a)}                    PDEPS: {{B}}
% Exit:  MODES: {(A,a),(B,a)}                    PDEPS: {∅,{A},{B}}
fillRectangle(A,C) :-
        AC1   A>=1,
        AC2   makeSquares(C),
        AC3   fillZone(⟨-1,A,1⟩,L,C,⟨⟩).   AC4
```

```
% Call:   makeSquares(A)
% Entry:  MODES: {(A,a)}                    PDEPS: {{A}}
% Exit:   MODES: {(A,a)}                    PDEPS: {∅,{A}}
makeSquares([]).   AC5
makeSquares([B|C]) :-
        AC6   B>0,
        AC7   makeSquares(C),
        AC8   makeDistinct(B,C).   AC9
```

fillRectangle/2		
AC_1	MODES	{(A,f),(C,a),(L,f)}
	PDEPS	{{C}}
AC_2	MODES	{(A,a),(C,a),(L,f)}
	PDEPS	{{A},{C}}
AC_3	MODES	{(A,a),(C,a),(L,f)}
	PDEPS	{∅,{A},{C}}
AC_4	MODES	{(A,a),(C,a),(L,a)}
	PDEPS	{∅,{A},{C},{L}}
makeSquares/1		
AC_5	MODES	∅
	PDEPS	{∅}
AC_6	MODES	{(B,a),(C,a)}
	PDEPS	{{B},{C}}
AC_7	MODES	{(B,a),(C,a)}
	PDEPS	{∅,{B},{C}}
AC_8	MODES	{(B,a),(C,a)}
	PDEPS	{∅,{B},{C}}
AC_9	MODES	{(B,a),(C,a)}
	PDEPS	{∅,{B},{C}}

The *makeSquares*/1 predicate is called with a tuple variable whose size is constrained, so the argument has mode a; however, the elements of the tuple are still free variables. The predicate constrains these elements (representing the sizes of the squares used to fill the rectangle) to be positive and distinct. In AC_1, the tuple element B has mode a although in the concrete execution the element is still free at that point. This imprecise mode is caused by the fact that B is a component of an argument with mode a, on which no structure information is available. If the analysis would maintain more precise structure information, i.e. the argument of the *makeSquares* predicate is a tuple with free elements, the more precise mode f could be derived for B.

Bibliography

[1] S. Abramsky and C. Hankin, editors. *Abstract Interpretation of Declarative Languages*. Ellis Horwood Series in Computers and their Applications. Ellis Horwood, Chichester, 1987.

[2] R. Bagnara, R. Giacobazzi, and G. Levi. Static analysis of CLP programs over numeric domains. In *Proceedings of the Workshop on Static Analysis*, pages 43–50, Bordeaux, Sept. 1992.

[3] R. Bagnara, R. Giacobazzi, and G. Levi. An application of constraint propagation to data-flow analysis. In *Proceedings of the 9th IEEE Conference on AI Applications (CAIA'93)*, 1993.

[4] N. Baker and H. Søndergaard. Definiteness analysis for CLP($\mathcal{R}$). *Australian Computer Science Communications*, 15(1):321–332, 1993.

[5] F. Berthier. Solving financial decision problems with CHIP. In *Proceedings of the 2nd Conference on Economics and Artificial Intelligence*, pages 233–238, Paris, June 1990.

[6] M. Bruynooghe. A practical framework for the abstract interpretation of logic programs. *Journal of Logic Programming*, 10(2):91–124, Feb. 1991.

[7] M. Bruynooghe and D. Boulanger. Abstract interpretation for (constraint) logic programming. Technical Report CW183, Department of Computer Science, Katholieke Universiteit Leuven, Nov. 1993. Appeared in Constraint programming, Eds. B. Mayoh, E. Tõugu and J. Penjam, NATO Advanced Science Series, Computers and System Sciences, Springer-Verlag, 1994.

[8] M. Bruynooghe and M. Codish. Freeness, sharing, linearity and correctness – all at once. In P. Cousot, M. Falaschi, G. Filé, and A. Rauzy, editors, *Proceedings of the 3rd International Workshop on Static Analysis*, number 724 in Lecture Notes in Computer Science, pages 153–164, Padova, Italy, Sept. 1993. Springer-Verlag.

[9] M. Bruynooghe, M. Codish, and A. Mulkers. A composite domain for freeness, sharing, and compoundness analysis of logic programs. Technical Report CW196, K.U.Leuven, Dept. of Computer Science, July 1994.

[10] M. Bruynooghe, V. Dumortier, and G. Janssens. Improving the efficiency of constraint logic programming languages by deriving specialized versions. In M. Richter and H. Boley, editors, *Proceedings of the International Workshop on Processing Declarative Knowledge (PDK'91)*, number 567 in Lecture Notes in Artificial Intelligence, pages 309–317, Kaiserslautern, July 1991. Springer-Verlag.

[11] M. Bruynooghe and G. Janssens. Towards a framework for the abstract interpretation of constraint logic programs. In A. Pettorossi, editor, *Proceedings of the Third Workshop*

on Meta-programming in Logic, number 649 in Lecture Notes in Computer Science, pages 294–307, Uppsala, Sweden, June 1992. Springer-Verlag.

[12] Y. Caseau and F. Laburthe. Improved CLP scheduling with task intervals. In P. Van Hentenryck, editor, *Proceedings of the 11th International Conference on Logic Programming*, pages 369–383, Italy, June 1994. MIT Press.

[13] M. Codish, D. Dams, G. Filé, and M. Bruynooghe. Freeness analysis for logic programs – and correctness ? In D. Warren, editor, *Proceedings of the 10th International Conference on Logic Programming*, pages 116–131, Budapest, Hungary, June 1993. MIT Press.

[14] M. Codish, D. Dams, and E. Yardeni. Bottom-up abstract interpretation of logic programs. Technical Report CS90-24, Department of Computer Science, Weizmann Institute of Science, Israel, Oct. 1990.

[15] M. Codish, D. Dams, and E. Yardeni. Derivation and safety of an abstract unification algorithm for aliasing analysis. In K. Furukawa, editor, *Proceedings of the 8th International Conference on Logic Programming*, pages 79–93, Paris, France, June 1991. MIT Press.

[16] M. Codish, A. Mulkers, M. Bruynooghe, M. García de la Banda, and M. Hermenegildo. Improving abstract interpretations by combining domains. In *Proceedings of the ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 194–205, Copenhagen, Denmark, June 1993. ACM New York.

[17] P. Codognet and G. Filé. Computations, abstractions and constraints in logic programs. In *Proceedings of the 4th International Conference on Programming Languages*, Apr. 1992.

[18] J. Cohen. Constraint logic programming languages. *Communications of the ACM*, 30(7):52–68, 1990.

[19] A. Colmerauer. An introduction to PROLOGIII. *Communications of the ACM*, 30(7):69–96, 1990.

[20] A. Cortesi and G. Filé. Abstract interpretation of logic programs : an abstract domain for groundness, sharing, freeness and compoundness analysis. In P. Hudak and N. Jones, editors, *Proceedings of the ACM Symposium on Partial Evaluation and Semantics Based Program Manipulation*, number 11 in SIGPLAN NOTICES 26, pages 52–61, Connecticut, USA, 1991. ACM Press.

[21] A. Cortesi, G. Filé, and W. Winsborough. Prop revisited : propositional formula as abstract domain for groundness analysis. In *Proceedings of the 6th Annual IEEE Symposium on Logic in Computer Science*, pages 322–327, Amsterdam, The Netherlands, July 1991. IEEE Computer Society Press.

[22] A. Cortesi, B. Le Charlier, and P. Van Hentenryck. Combinations of abstract domains for logic programming. In *Proceedings of the 21st Annual Symposium on Principles Of Programming Languages*, Portland, Oregon, Jan. 1994.

[23] P. Cousot and R. Cousot. Abstract interpretation : a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, Jan. 1977.

[24] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proceedings of the 6th ACM Symposium on Principles of Programming Languages*, pages 269–282, San Antonio, Texas, Jan. 1979.

[25] P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *Journal of Logic Programming*, 13(2&3):103–179, 1992.

[26] P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic Computation*, 2(4):511–547, 1992.

[27] P. Cousot and R. Cousot. Comparing the Galois connection with widening/narrowing approaches to abstract interpretation. In M. Bruynooghe and M. Wirsing, editors, *Proceedings of the 4th International Symposium on Programming Language Implementation and Logic Programming*, number 631 in Lecture Notes in Computer Science, pages 269–295, Leuven, Belgium, Aug. 1992.

[28] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th ACM Symposium on Principles of Programming Languages*, pages 84–97, Tucson, Arizona, Jan. 1978.

[29] D. De Schreye and M. Bruynooghe. On the transformation of logic programs with instantiation based computation rules. *Journal of Symbolic Computation*, 7:125–154, 1989.

[30] S. K. Debray. Unfold/fold transformations and loop optimization of logic programs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, number 7 in SIGPLAN Notices 23, pages 297–307, 1988.

[31] S. K. Debray. Static inference of modes and data dependencies in logic programs. *ACM Transactions on Programming Languages and Systems*, 11(3):418–450, 1989.

[32] S. K. Debray. Efficient dataflow analysis of logic programs. *Journal of the ACM*, 39(4):949–984, Oct. 1992.

[33] S. K. Debray, editor. Journal of Logic Programming, special issue : Abstract interpretation. 13(1–2), July 1992.

[34] M. Dincbas, H. Simonis, and P. Van Hentenryck. Solving a cutting-stock problem in Constraint Logic Programming. In R. A. Kowalski and K. A. Bowen, editors, *Proceedings of the 5th International Conference on Logic Programming*, pages 42–58, Seattle, June 1988. MIT Press.

[35] M. Dincbas, P. Van Hentenryck, H. Simonis, and A. Aggoun. The Constraint Logic Programming Language CHIP. In *Proceedings of the 2nd International Conference on Fifth Generation Computer Systems*, pages 249–264, Tokyo, Japan, Nov. 1988.

[36] V. Dumortier and G. Janssens. Towards a practical full mode inference system for CLP(H,N). In P. Van Hentenryck, editor, *Proceedings of the 11th International Conference on Logic Programming*, pages 569–583, Italy, June 1994. MIT Press.

[37] V. Dumortier, G. Janssens, and M. Bruynooghe. Detection of Free Variables in the Presence of Numeric Constraints. In *Proceedings of the JICSLP'92 Post-conference workshop on CLP*, pages 105–118, Washington, Nov. 1992.

[38] V. Dumortier, G. Janssens, M. Bruynooghe, and M. Codish. Freeness analysis in the presence of numerical constraints. In D. S. Warren, editor, *Proceedings of the 10th International Conference on Logic Programming*, pages 100–115, Budapest, Hungary, June 1993. MIT Press.

[39] V. Englebert, B. Le Charlier, D. Roland, and P. Van Hentenryck. Generic abstract interpretation algorithms for Prolog : two optimization techniques and their experimental evaluation. In M. Bruynooghe and M. Wirsing, editors, *Proceedings of the 4th International Symposium on Programming Language Implementation and Logic Programming*, number 631 in Lecture Notes in Computer Science, pages 311–325, Leuven, Belgium, Aug. 1992. Springer-Verlag. Also in *Software Practice and Experience*, 23(4):419–460, 1993.

[40] J. Gallagher and D. de Waal. Fast and precise regular approximations of logic programs. In P. Van Hentenryck, editor, *Proceedings of the 11th International Conference on Logic Programming*, pages 599–613, Italy, June 1994. MIT Press.

[41] M. García de la Banda. *Independence, Global Analysis, and Parallelism in Dynamically Scheduled Constraint Logic Programming*. PhD thesis, Universidad Politécnica de Madrid (UPM), July 1994.

[42] M. García de la Banda and M. Hermenegildo. Some Considerations on the Compile-Time Analysis of Constraint Logic Programs. In *Jornadas Nacionales de Programación Declarativa*, pages 97–117, Malaga, Spain, Oct. 1991. U. Malaga.

[43] M. García de la Banda and M. Hermenegildo. A Practical Approach to the Global Analysis of CLP Programs. In D. Miller, editor, *Proceedings of the 10th International Logic Programming Symposium*, pages 437–455, Vancouver, Canada, Oct. 1993. MIT Press.

[44] M. García de la Banda, M. Hermenegildo, and K. Marriott. Independence in Constraint Logic Programs. In D. Miller, editor, *Proceedings of the 10th International Logic Programming Symposium*, Vancouver, Canada, Oct. 1993. MIT Press.

[45] R. Giacobazzi, S.K.Debray, and G. Levi. A generalized semantics for constraint logic programs. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 581–591, Tokyo, Japan, June 1992.

[46] R. Giacobazzi, S.K.Debray, and G. Levi. Generalized semantics and abstract interpretation for constraint logic programs. Preliminary Report, Apr. 1993.

[47] F. Giannotti and M. Hermenegildo. A technique for recursive invariance detection and selective program specialization. In J. Małuszyński and M. Wirsing, editors, *Proceedings of the 3rd International Symposium on Programming Language Implementation and Logic Programming*, number 528 in Lecture Notes in Computer Science, pages 323–334, Passau, Germany, Aug. 1991. Springer-Verlag.

[48] M. Hanus. Analysis of nonlinear constraints in CLP(R). In D. S. Warren, editor, *Proceedings of the 10th International Conference on Logic Programming*, pages 83–99, Budapest, Hungary, June 1993. MIT Press.

[49] N. Heintze, S. Michaylov, and P. Stuckey. CLP($\mathcal{R}$) and some electrical engineering problems. *Journal of Automated Reasoning*, 9:231–260, 1992. Also in the *Proceedings of the 4th International Conference on Logic Programming*, pages 675–703, 1987.

[50] E. Horowitz and S. Sahni. *Fundamentals of Data Structures in Pascal.* Pitman Publishing Limited, 128 long Acre, London WC2E 9 AN, 1984.

[51] T. Huyhn and C. Lassez. An expert decision-support system for option-based investment. *Computer Mathematics with Applications*, 20(9&10):1–14, 1990.

[52] J.-L. Imbert. Variable elimination for generalized linear constraints. In D. S. Warren, editor, *Proceedings of the 10th International Conference on Logic Programming*, pages 499–516, Budapest, Hungary, June 1993.

[53] D. Jacobs and A. Langen. Accurate and efficient approximation of variable aliasing in logic programs. In E. Lusk and R. Overbeek, editors, *Proceedings of the North American Conference on Logic Programming*, pages 154–165, Cleveland, Ohio, Oct. 1989. MIT Press.

[54] J. Jaffar and J.-L. Lassez. Constraint logic programming. In *Proceedings of the 14th ACM Symposium on Principles of Programming Languages*, pages 111–119, Munich, Germany, Jan. 1987. ACM New York.

[55] J. Jaffar and M. J. Maher. Constraint Logic Programming : a survey. *Journal of Logic Programming*, 19/20:503–581, May/July 1994.

[56] J. Jaffar, S. Michaylov, P. Stuckey, and R. Yap. An abstract machine for CLP(R). In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 128–139, San Francisco, June 1992.

[57] J. Jaffar, S. Michaylov, P. J. Stuckey, and R. H. Yap. The CLP(R) language and system. *ACM Transactions on Programming Languages and Systems*, 14(3):339–395, July 1992.

[58] S. Janson and S. Haridi. Programming paradigms of the Andorra Kernel Language. In V. Saraswat and K. Ueda, editors, *Proceedings of the 1991 International Logic Programming Symposium*, pages 167–183, San Diego, USA, Oct. 1991. MIT Press.

[59] G. Janssens. *Deriving Run Time Properties of Logic Programs by means of Abstract Interpretation.* PhD thesis, Department of Computer Science, Katholieke Universiteit Leuven, Mar. 1990.

[60] G. Janssens, M. Bruynooghe, and V. Englebert. Abstracting numerical values in CLP(H,N). In M. Hermenegildo and J. Penjam, editors, *Proceedings of the 6th International Symposium on Programming Language Implementation and Logic Programming (PLILP'94)*, pages 400–414. Springer-Verlag, Sept. 1994.

[61] G. Janssens and W. Simoens. On the implementation of abstract interpretation systems for (constraint) logic programs. In P. A. Fritzson, editor, *Proceedings of the 5th International Conference on Compiler Construction*, number 786 in Lecture Notes in Computer Science, pages 172–187, Edinburgh, Apr. 1994. Springer-Verlag.

[62] N. Jørgensen. *Abstract interpretation of constraint logic programs.* PhD thesis, Roskilde University, Computer Science, July 1992.

[63] N. Jørgensen, K. Marriott, and S. Michaylov. Some global compile-time optimizations for CLP(R). In V. Saraswat and K. Ueda, editors, *Proceedings of the 1991 International Symposium on Logic Programming*, pages 420–434, San Diego, USA, Oct. 1991. MIT Press.

[64] A. King and P. Soper. Depth-k sharing and freeness. In P. Van Hentenryck, editor, *Proceedings of the 11th International Conference on Logic Programming*, pages 553–568, Italy, June 1994. MIT Press.

[65] R. A. Kowalski. *Logic for problem solving*. North Holland, New York, 1979.

[66] R. L. Kruse. *Data structures and program design*. Prentice-Hall software series. Prentice-Hall, Inc., Eglewood Cliffs, New Jersey 07632, 1984.

[67] C. Lassez, K. McAloon, and R. Yap. Constraint logic programming and options trading. *IEEE Expert, Special Issue on Financial Software*, 2(3):42–50, Aug. 1987.

[68] J.-L. Lassez, M. Maher, and K. Marriott. Unification revisited. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 587–625. Morgan Kaufmann, Los Altos, Ca., 1988.

[69] J.-L. Lassez and K. McAloon. A canonical form for generalised linear constraints. *Journal of Symbolic Computation*, 13(1):1–24, Jan. 1992.

[70] B. Le Charlier, K. Musumbu, and P. Van Hentenryck. A generic abstract interpretation algorithm and its complexity analysis (extended abstract). In K. Furukawa, editor, *Proceedings of the 8th International Conference on Logic Programming*, pages 64–78, Paris, June 1991. MIT Press.

[71] B. Le Charlier and P. Van Hentenryck. Reexecution in abstract interpretation of Prolog. In K. Apt, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 750–764, Washington, Nov. 1992. MIT Press.

[72] B. Le Charlier and P. Van Hentenryck. Experimental evaluation of a generic abstract interpretation algorithm for prolog. In *ACM Transactions on Programming Languages and Systems (TOPLAS)*, Jan. 1994.

[73] J. Lee and M. Van Emden. Interval computation as deduction in CHIP. *Journal of Logic Programming*, 16(3&4):255–276, July/Aug. 1993.

[74] J. W. Lloyd. *Foundations of Logic Programming*. Springer Series : Symbolic Computation - Artificial Intelligence. Springer-Verlag, second, extended edition, 1987.

[75] A. D. Macdonald, P. J. Stuckey, and R. H. C. Yap. Redundancy of variables in CLP(R). In D. Miller, editor, *Proceedings of the 10th International Logic Programming Symposium*, pages 75–93, Vancouver, Canada, Oct. 1993. MIT Press.

[76] A. Mackworth. *Handbook of AI*, chapter Constraint Satisfaction, pages 205–211. 1988.

[77] A. Marien, G. Janssens, A. Mulkers, and M. Bruynooghe. The impact of abstract interpretation : an experiment in code generation. In G. Levi and M. Martelli, editors, *Proceedings of the 6th International Conference on Logic Programming*, pages 33–47, Lisbon, Portugal, June 1989. MIT Press.

[78] K. Marriott. Frameworks for abstract interpretation. *Acta Informatica*, 30:103–129, 1993.

[79] K. Marriott, M. García de la Banda, and M. Hermenegildo. Analyzing logic programs with dynamic scheduling. In *Proceedings of the 20th Annual ACM Conference on Principles of Programming Languages*. ACM, Jan. 1994.

[80] K. Marriott and H. Søndergaard. Bottom-up abstract interpretation of logic programs. In R. A. Kowalski and K. A. Bowen, editors, *Proceedings of the 5th International Conference and Symposium on Logic Programming*, pages 733–748, Seattle, Aug. 1988. MIT Press.

[81] K. Marriott and H. Søndergaard. Notes for a tutorial on abstract interpretation of logic programs. In E. L. Lusk and R. A. Overbeek, editors, *Proceedings of the North American Conference on Logic Programming*, Cleveland, Ohio, Oct. 1989. MIT Press.

[82] K. Marriott and H. Søndergaard. Semantics-based dataflow analysis of logic programs. *Information Processing*, pages 601–606, Apr. 1989.

[83] K. Marriott and H. Søndergaard. Analysis of constraint logic programs. In S. Debray and M. Hermenegildo, editors, *Proceedings of the 1990 North American Conference on Logic Programming*, pages 531–547, Austin, Texas, Oct. 1990. MIT Press.

[84] K. Marriott, H. Søndergaard, and N. Jones. Denotational abstract interpretation of logic programs. Technical Report 92/20, Dept. of Computer Science, University of Melbourne, Melbourne, 1992.

[85] K. Marriott, H. Søndergaard, P. J. Stuckey, and R. H. Yap. Optimizing compilation for CLP($\mathcal{R}$). In *Proceedings of the 17th Annual Computer Science Conference*, Christchurch, New Zealand, Jan. 1994.

[86] K. Marriott and P. Stuckey. The 3 R's of optimizing constraint logic programs : Refinement, Removal and Reordering. In *Proceedings of the 20th ACM Symposium on Principles of Programming Languages*, pages 334–344, Charleston, South Carolina, Jan. 1993.

[87] K. Marriott and P. J. Stuckey. Approximating interaction between linear arithmetic constraints. To appear in the Proceedings of ILPS94, Nov. 1994.

[88] A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4:258–282, 1982.

[89] C. S. Mellish. Abstract interpretation of Prolog programs. In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declarative Languages*, Ellis Horwood Series in Computers and their Applications, pages 181–198. Ellis Horwood, Chichester, 1987.

[90] S. S. Muchnick and N. D. Jones. *Program Flow Analysis : theory and Applications*. Prentice-Hall, Inc., 1981.

[91] A. Mulkers. *Deriving Live Data Structures in Logic Programs by Means of Abstract Interpretation*. PhD thesis, Department of Computer Science, Katholieke Universiteit Leuven, Dec. 1991.

[92] A. Mulkers, W. Winsborough, and M. Bruynooghe. Analysis of shared data structures for compile-time garbage collection in logic programs. In D. H. D. Warren and P. Szeredi, editors, *Proceedings of the 7th International Conference on Logic Programming*, pages 747–762, Jerusalem, June 1990. MIT Press.

[93] K. Musumbu. *Interprétation Abstraite de Programmes Prolog*. PhD thesis, University of Namur (Belgium), Sept. 1990.

[94] K. Muthukumar and M. Hermenegildo. Deriving a fixpoint computation algorithm for top-down abstract interpretation of logic programs. Technical Report ACT-DC-153-90, Microelectronics and Computer Technology Corporation (MCC), Austin, TX 78759, Apr. 1990.

[95] K. Muthukumar and M. Hermenegildo. Combined determination of sharing and freeness of program variables through abstract interpretation. In K. Furukawa, editor, *Proceedings of the 8th International Conference on Logic Programming*, pages 49–63, Paris, France, June 1991. MIT Press.

[96] K. Muthukumar and M. Hermenegildo. Compile-time Derivation of Variable Dependency Using Abstract Interpretation. *Journal of Logic Programming*, 13(2&3):315–347, July 1992.

[97] U. Nilsson. Systematic semantic approximations of logic programs. In P. Deransart and J. Małuszyński, editors, *Proceedings of the International Workshop on Programming Language Implementation and Logic Programming*, number 456 in Lecture Notes in Computer Science, pages 293–306, Linköping, Sweden, Aug. 1990. Springer-Verlag.

[98] U. Nilsson. Abstract interpretation : a kind of magic. In J. Małuszyński and M. Wirsing, editors, *Proceedings of the 3rd International Symposium on Programming Language Implementation and Logic Programming*, number 528 in Lecture Notes in Computer Science, Passau, Germany, Aug. 1991. Springer-Verlag.

[99] D. A. Plaisted. The occur-check problem in Prolog. *New Generation Computing*, 2(4):309–322, 1984. Also in : *Proceedings of the International Symposium on Logic Programming*, pages 272–280, Atlantic City, 1984. IEEE Computer Society Press.

[100] PrologIA, Parc Technologique de Luminy - Case 919, 13288 Marseille Cedex 09, France. *Prolog III Version 1.3 Reference Manual*, Dec. 1991.

[101] J. A. Robinson. A machine-oriented logic based on the resolution principle. *JACM*, 12(1):23–41, 1965.

[102] V. A. Saraswat. *Concurrent constraint programming languages*. PhD thesis, CMU, School of CS, Pittsburgh, Jan. 1989.

[103] R. Skuppin and T. Bückle. CLP and spacecraft attitude control. In *Proceedings of the JICSLP Workshop on Constraint Logic Programming*, pages 45–54, Washington, Nov. 1992.

[104] H. Søndergaard. An application of abstract interpretation of logic programs : occur check reduction. In B. Robinet and R. Wilhelm, editors, *Proceedings of the European Symposium on Programming*, number 213 in Lecture Notes in Computer Science, pages 327–338. Springer-Verlag, Saarbrücken, Mar. 1986.

[105] G. L. Steele. *The definition and implementation of a computer programming language based on constraints*. PhD thesis, MIT, Dept. of EE and CS, Cambridge, Mass., Aug. 1980.

[106] I. E. Sutherland. Sketchpad : a man-machine graphical communication system. Technical Report 296, MIT Lincoln Laboratory, Cambridge, MA, 1964.

[107] A. Taylor. LIPS on a MIPS : results from a Prolog compiler for a RISC. In D. H. D. Warren and P. Szeredi, editors, *Proceedings of the 7th International Conference on Logic Programming*, pages 174–185, Jerusalem, June 1990. MIT Press.

[108] A. Taylor. *High Performance Prolog Implementation.* PhD thesis, University of Sydney, June 1991.

[109] P. Van Hentenryck. Constraint logic programming. *Knowledge Engineering Review*, 6:151–194, 1991.

[110] P. Van Roy and A. M. Despain. The benefits of global dataflow analysis for an optimizing Prolog compiler. In E. L. Lusk and R. A. Overbeek, editors, *Proceedings of the North American Conference on Logic Programming*, pages 501–515, Cleveland, Ohio, Oct. 1989. MIT Press.

[111] K. Verschaetse and D. De Schreye. Derivation of linear size relations by abstract interpretation. In *Proceedings of the 4th International Symposium on Programming Language Implementation and Logic Programming*, number 631 in Lecture Notes in Computer Science, pages 296–310, Leuven, Belgium, Aug. 1992. Springer-Verlag.

[112] R. Warren, M. Hermenegildo, and S. K. Debray. On the practicality of global flow analysis of logic programs. In R. A. Kowalski and K. A. Bowen, editors, *Proceedings of the 5th International Conference and Symposium on Logic Programming*, pages 684–699, Seattle, Aug. 1988. MIT Press.

Vrijheids- en aanverwante analyses van constraint logische programma's met behulp van abstracte interpretatie

Veroniek Dumortier

Departement Computerwetenschappen, K.U.Leuven

SAMENVATTING

Deze thesis handelt over het afleiden van uitvoeringseigenschappen van constraint logische programma's. Constraint logisch programmeren is expressiever dan logisch programmeren door het toevoegen van constraints. Dit komt echter vaak ten koste van performantie. Voor bepaalde klassen van vraagstellingen aan een programma is de volledige kracht van de constraint oplossers niet nodig. In zulke gevallen kan een vertaler uitvoeringseigenschappen gebruiken om gespecialiseerde en meer efficiënte code te genereren.

Het afleiden van uitvoeringseigenschappen gebeurt via abstracte interpretatie. Dit is een methode voor interprocedurale programma-analyse. Een bestaand raamwerk voor de analyse van (constraint) logische programma's wordt gebruikt. Dit laat de ontwerper van een specifieke analyse toe zich te concentreren op de toepassingsafhankelijke componenten en correctheidsvoorwaarden. Ook kunnen implementaties van de domein-onafhankelijke analyse-procedure hergebruikt worden.

Binnen het raamwerk wordt een analyse ontwikkeld voor het detecteren van mogelijke constraint interactie. Deze analyse leidt af welke variabelen optreden als *vrijheidsgraden* met betrekking tot de oplosbaarheid van de constraints waarin ze voorkomen. De afgeleide informatie maakt verschillende programma-optimalisaties mogelijk, zoals het herordenen of het parallel uitvoeren van constraints en/of procedure-oproepen. De analyse spitst zich hoofdzakelijk toe op constraint logische programmeertalen die rekenen over het term domein (d.i. het domein bij logische programmeertalen) en over een numeriek domein.

De efficiëntie van de basisanalyse kan op twee manieren verbeterd worden : (1) door enkel *minimale* informatie af te leiden, en (2) door gebruik te maken van informatie over het uniek beperkt zijn van variabelen. De combinatie van deze optimalisaties leidt tot een praktische en volledige analyse voor het detecteren of variabelen niet of uniek beperkt zijn. Daarnaast worden een aantal uitbreidingen voorgesteld, (zoals het behandelen van niet-genormaliseerde programma's, het integreren van type informatie, enz.), die de kracht van de analyse verder verhogen. Het verkregen systeem vormt de basis voor een brede waaier van programma-optimalisaties.

Inhoudsopgave

1 Inleiding

Constraint Logisch Programmeren (CLP)[1] [6] combineert de voordelen van Constraint Programmeren (CP) en Logisch Programmeren (LP). CLP is een uitbreiding van LP waarbij het unificatie-mechanisme vervangen wordt door het meer algemene mechanisme van oplossen van constraints, bijvoorbeeld het oplossen van vergelijkingen en ongelijkheden over het domein van de reële getallen; LP is een speciaal geval met als enige constraints gelijkheden tussen Herbrand termen. De uitdrukkingskracht en multi-directionaliteit worden aanzienlijk verhoogd, wat leidt tot een ruimer toepassingsgebied. CLP talen zijn ideaal voor de snelle ontwikkeling van prototypes. Voor sommige applicaties is de efficiëntie van een CLP programma vergelijkbaar met de performantie van een programma geschreven in een andere (conventionele) taal, vooral wanneer de constraints effectief kunnen gebruikt worden voor het snoeien van de zoekruimte. Vaak wordt de verhoogde expressiviteit echter duur betaald. Om de performantie van CLP systemen te verhogen moet het toepassen van algemene constraint oplossers ("solvers") vermeden worden indien de volledige kracht van deze oplossers niet nodig is. Bijvoorbeeld, indien alle variabelen in een numerieke ongelijkheid een waarde hebben wanneer de ongelijkheid wordt geëvalueerd, dan kan de ongelijkheid omgezet worden in een eenvoudige test. Ook optimalisaties op een hoger niveau, zoals het herordenen van constraints, kunnen een drastische verhoging van de efficiëntie teweegbrengen. Om dergelijke optimalisaties te kunnen doorvoeren is een globale analyse van CLP programma's noodzakelijk. Een reeds lang erkende techniek hiervoor is abstracte interpretatie. Via de analyse worden eigenschappen afgeleid die gelden tijdens de uitvoering van een programma. Die eigenschappen kunnen dan gebruikt worden door een vertaler om meer efficiënte code te genereren.

In deze thesis wordt een bestaand raamwerk ("framework") voor abstracte interpretatie van LP talen gebruikt, meer bepaald het raamwerk van Bruynooghe [1]. Dit raamwerk werd reeds aangepast voor het analyseren van CLP talen [5]. De beschrijving in [5] wordt in grote lijnen gevolgd, behalve dat in deze thesis meer nadruk gelegd wordt op de *primitieve* abstracte operaties [4, 3] en dat de eindigheid van de analyse gegarandeerd wordt in termen van een verwijdingsoperatie ("widening operation"). De aanpassing naar CLP bestaat er hoofdzakelijk in de LP noties *term domein*, *substitutie* en *unificatie* te vervangen door de CLP noties *constraint domein*, *constraint* en *conjunctie* van een constraint met de constraints verzameld tot op een zeker punt tijdens de uitvoering. Het raamwerk definieert de globale analyse-procedure, waarvan reeds efficiënte implementaties bestaan. Het ontwikkelen van een specifieke analyse binnen het raamwerk komt neer op het ontwerpen van de toepassingsafhankelijke delen. Deze moeten voldoen aan bepaalde (voor CLP aangepaste) voorwaarden die de correctheid en eindigheid van de volledige analyse garanderen.

Binnen het aangepast raamwerk wordt een specifieke analyse ontwikkeld voor het afleiden van mogelijke interactie tussen constraints. De analyse is toegespitst op CLP talen die het LP term domein en een numeriek constraint domein (meer bepaald het domein van de rationale of de reële getallen) omvatten. De analyse bepaalt welke variabelen optreden

[1] De vertaling "logisch programmeren met beperkingen" wordt bewust vermeden : deze vertaling zou de indruk kunnen wekken dat constraint logisch programmeren minder krachtig is dan logisch programmeren, terwijl net het omgekeerde geldt.

als *vrijheidsgraden* met betrekking tot het oplosbaar zijn van de constraints waarin ze voorkomen. Twee orthogonale optimalisaties van de basisanalyse worden voorgesteld : (1) het enkel bijhouden van *minimale* informatie en (2) het benutten van "definiteness" informatie (die aangeeft of variabelen al dan niet beperkt zijn tot een unieke waarde) [5]. De combinatie van beide resulteert in een praktisch en volledig mode-systeem dat op elk punt beschrijft welke variabelen niet of uniek beperkt zijn.

Verder worden een aantal uitbreidingen van de basisanalyse aangegeven, zoals het toevoegen van type-informatie of het behandelen van andere constraint domeinen.

2 Beginselen van constraint logisch programmeren

Constraint Logisch Programmeren (CLP) vervangt het unificatie-mechanisme in LP door het oplossen van constraints over bepaalde domeinen. In deze thesis gaat de aandacht in eerste instantie naar CLP(H,N) talen, die twee constraint domeinen omvatten : het LP term domein, H, en een numeriek domein N (het domein van de rationale of reële getallen). De semantiek van CLP talen komt grotendeels overeen met die van LP talen, behalve dat unificeerbaarheid vervangen wordt door oplosbaarheid van constraints. Voor meer details verwijzen we naar [6].

Een *primitieve* constraint is een beperking tussen termen. In het CLP(H,N) geval is dit een gelijkheid tussen Herbrand (LP) termen – een primitieve unificatie constraint genoemd – of een gelijkheid ($=$), verschil ($\neq$) of ongelijkheid ($>, \geq$) tussen numerieke termen – een primitieve numerieke constraint. Een constraint is een conjunctie ($\wedge$) van primitieve constraints. Hier wordt ook een onderscheid gemaakt tussen een unificatie constraint (conjunctie van primitieve unificatie constraints), een numerieke constraint (conjunctie van primitieve numerieke constraints) en een gemengde constraint (conjunctie van primitieve unificatie en/of numerieke constraints). De verzameling van alle constraints wordt genoteerd als *Cons*.

De conjuncties van respectievelijk alle primitieve unificatie constraints en alle primitieve numerieke constraints die geïmpliceerd worden door een constraint C, worden genoteerd als $unif^*(C)$ en $num^*(C)$. Merk op dat een numerieke constraint een unificatie constraint kan impliceren en vice versa. Bijvoorbeeld, $X - Y = 0$ impliceert $X = Y$.

Een constraint C kan steeds getransformeerd worden naar een equivalente constraint in *opgeloste vorm* ("solved form"), genoteerd $sform(C)$. De opgeloste vorm van een unificatie constraint is een conjunctie

$$X_1 = t_1 \wedge \ldots \wedge X_n = t_n$$

waarbij de X_i verschillende variabelen zijn en de t_i Herbrand termen, en de X_i komen niet voor in de t_i. Deze vorm kan ook gezien worden als een substitutie $\theta = \{X_1 \leftarrow t_1, \ldots, X_n \leftarrow t_n\}$. De variabelen in de t_i zijn de parameters van de opgeloste vorm; de opgeloste vorm is uniek op de keuze van parameters na. Voor een numerieke constraint (althans voor een conjunctie van lineaire gelijkheden) bestaat een gelijkaardige opgeloste vorm, waarbij de t_i numerieke termen zijn[2]. De opgeloste vorm van een gemengde constraint C bestaat uit

[2] Een primitieve numerieke gelijkheid wordt echter meestal geschreven in de vorm $a_1X_1 + \ldots + a_nX_n = b$ in plaats van $X_1 = t_1$.

de opgeloste vormen van het unificatie gedeelte $unif^*(C)$ en van het numerieke gedeelte $num^*(C)$. De primitieve constraints in de opgeloste vorm worden geïmpliceerd door de oorspronkelijke constraint. De opgeloste vorm maakt informatie die verborgen was in de oorspronkelijke vorm van de constraint meer expliciet.

De combinatie van twee constraint domeinen in CLP(H,N) leidt tot een typering van de variabelen. Beschouw een constraint C. Een variabele X in C is een *Herbrand* variabele indien C een primitieve constraint $X = t$ impliceert waarbij t geen numerieke term (en dus ook geen variabele) is. De variabele X is *numeriek* indien hij voorkomt in een primitieve numerieke constraint in C of indien hij gebonden is met een numerieke variabele Y via een (geïmpliceerde) primitieve constraint $X = Y$ of $Y = X$. Anders is X *niet-getypeerd* (dus X kan dan enkel voorkomen in primitieve constraints van de vorm $X = Y$ of $Y = X$ waarbij Y niet-getypeerd is).

3 Abstracte interpretatie

Abstracte interpretatie is een methode voor globale of interprocedurale programma-analyse. De analyse kan beschouwd worden als het uitvoeren van het programma over een domein van data beschrijvingen of benaderingen (*abstracte* data) in plaats van over de concrete data. Het domein van beschrijvingen wordt het *abstract domein* genoemd. Operaties op concrete data worden vervangen door *abstracte operaties* op de abstracte data. Deze abstracte operaties moeten het effect van hun concrete tegenhangers nabootsen opdat de analyse correct zou zijn.
Een bekend voorbeeld uit de wiskunde is de teken-analyse. Het doel is het teken te bepalen van het resultaat van een numerieke operatie, zoals de vermenigvuldiging van twee getallen. In plaats van voor alle concrete getallen het produkt te berekenen en te analyseren, worden getallen benaderd door hun teken en wordt een abstracte vermenenigvuldiging $*^a$ over het teken-domein gedefinieerd, die de concrete vermenigvuldiging nabootst. Op die manier moeten slechts een eindig aantal gevallen beschouwd worden. Veronderstel dat de concrete data gehele getallen zijn. Het abstract domein, in dit geval de verzameling van mogelijke tekens, is dan $\{+, -, 0\}$. De *concretisatiefunctie* γ associeert elk teken met de verzameling van getallen die dat teken hebben, bijvoorbeeld $\gamma(+) = \mathbb{Z}_0^+$. De *abstractiefunctie* α associeert elk getal met zijn teken, bijvoorbeeld $\alpha(-5) = -$. De abstracte vermenigvuldiging $*^a$ is gedefinieerd aan de hand van de onderstaande tabel.

$*^a$	$-$	0	$+$
$-$	$+$	0	$-$
0	0	0	0
$+$	$-$	0	$+$

$*^a$ is een correcte tegenhanger van $*$ vermits $\forall z_1, z_2 \in \mathbb{Z}.: \alpha(z_1 * z_2) = \alpha(z_1) *^a \alpha(z_2)$.

Verschillende raamwerken werden ontwikkeld voor de abstracte interpretatie van LP talen. Een raamwerk is een generische constructie. Het omvat de globale analyse-procedure en formuleert condities op de toepassingsafhankelijke componenten waardoor de correctheid en eindigheid van de volledige analyse gegarandeerd worden. Een specifieke analyse wordt verkregen door de toepassingsafhankelijke domeinen en operaties in te vullen.

Het raamwerk dat in deze thesis wordt gebruikt is dat van Bruynooghe [1]. Dit is een zogenaamd "top-down" raamwerk. Vertrekkend van een beschrijving van verwachte vraagstellingen aan een programma worden eigenschappen afgeleid van de programma-toestand op elk programmapunt. *Programmapunten* zijn de punten voor en na een constraint of procedure-oproep. Elke beschrijving wordt beperkt tot de variabelen die zichtbaar zijn op het programmapunt waarmee de beschrijving geassocieerd wordt (d.i. de variabelen in de programma-regel of vraagstelling); deze verzameling van variabelen wordt het *domein* van de beschrijving genoemd. De beperking is essentieel voor het behandelen van recursieve procedures. De analyse wordt georganiseerd rond een EN/OF grafe die een verzameling van (mogelijk oneindige) concrete EN/OF bomen beschrijft. Elke EN/OF boom stelt een concrete uitvoering van het programma voor met betrekking tot één van de beschreven vraagstellingen. Tijdens abstracte interpretatie kunnen verschillende *versies* van een procedure ontdekt worden; deze ontstaan door procedure-oproepen die syntactisch verschillen of die een andere oproep-beschrijving hebben.
Hoewel het raamwerk oorspronkelijk ontwikkeld werd voor de analyse van logische programma's, kan het vrij gemakkelijk aangepast worden voor de analyse van constraint logische programma's, zoals hierna wordt aangegeven. We volgen hierbij in grote lijnen de beschrijving in [5]. Er wordt echter meer nadruk gelegd op de *primitieve* abstracte operaties en de eindigheid van de analyse wordt gegarandeerd in termen van een verwijdingsoperatie.

3.1 Concreet en abstract domein

Elementen van het concreet domein Con^c zijn verzamelingen van constraints. Deze worden aangeduid met CS in de rest van de tekst. De orde relatie op het concreet domein is $\subseteq$.
Elementen van het abstract domein Con^a worden *abstracte constraints* genoemd en worden aangeduid met AC in de rest van de tekst. De abstracte constraints op de programmapunten vóór en na een constraint of procedure-oproep worden respectievelijk de *abstracte oproep-constraint* en de *abstracte terugkeer-constraint* genoemd. Elke abstracte constraint beschrijft een verzameling concrete constraints die op een programmapunt tijdens de uitvoering kunnen voorkomen. Het abstract domein Con^a kan opgesplitst worden in een aantal deeldomeinen Con^a_D, waarbij Con^a_D de abstracte constraints met hetzelfde domein D (verzameling van variabelen) bevat. Ook Con^c kan opgesplitst worden in verschillende Con^c_D; γ en α leggen dan het verband tussen elementen van Con^a_D en Con^c_D. Elke Con^a_D moet de volgende algebraïsche structuur hebben :

1. een orde relatie $\leq^a$ zodat $\forall AC_1, AC_2 \in Con^a_D : AC_1 \leq^a AC_2 \Rightarrow \gamma(AC_1) \leq^c \gamma(AC_2)$. Deze orde relatie induceert een equivalentie relatie $\equiv^a$.
2. een bovengrens operatie upp ("upperbound") zodat $\forall AC_1, AC_2 \in Con^a_D : AC_1 \leq^a upp(AC_1, AC_2)$ & $AC_2 \leq^a upp(AC_1, AC_2)$.
3. een maximaal element $AC_{max} \in Con^a_D$ zodat $\forall AC \in Con^a_D : AC \leq^a AC_{max}$ en $\gamma(AC_{max}) = Cons_D$.
4. een minimaal element $\bot \in Con^a_D$ zodat $\forall AC \in Con^a_D : \bot \leq^a AC$ en $\gamma(\bot) = \emptyset$.
5. een verwijdingsoperatie ("widening operation") W zodat
   - $\forall AC_1, AC_2 \in Con^a_D : W(AC_1, AC_2) \geq^a AC_1$ en $W(AC_1, AC_2) \geq^a AC_2$ en
   - voor alle stijgende ketens $AC_1 \leq^a AC_2 \leq^a \ldots$ met $AC_i \in Con^a_D$ mag de stijgende keten gedefinieerd door $AC'_1 = AC_1, \ldots, AC'_{i+1} = W(AC'_i, AC_{i+1}), \ldots$ niet strikt stijgend zijn.

Merk op dat als Con^a_D eindig is of geen oneindige stijgende keten bevat voor $\leq^a$, dan kan W gedefinieerd worden als *upp*.

De laatste vereiste is een standaard vereiste voor het garanderen van de eindigheid van de analyse en vervangt de meer specifieke vereiste in het originele raamwerk (deze aanpassing is niet CLP gebonden). Mits strengere voorwaarden[3] kunnen alle bovenstaande vereisten uitgedrukt worden in α in plaats van in γ. Dit is handig indien de definitie van α meer voor de hand ligt dan de definitie van γ (zie b.v. Sectie 4).

3.2 Abstracte operaties

De abstracte operaties in het originele raamwerk zijn *abstracte unificatie*, *procedure-oproep* en *procedure-terugkeer*. In het geval van CLP worden twee wijzigingen aangebracht. Abstracte unificatie wordt vervangen door *abstracte interpretatie van een constraint*. Daarnaast worden twee *primitieve* abstracte operaties ingevoerd, namelijk *abstracte conjunctie* en *abstracte projectie*. De andere abstracte operaties worden in termen daarvan gedefinieerd.
Om de definities te vereenvoudigen nemen we aan dat bronprogramma's genormaliseerd zijn. Dit betekent dat alle constraints expliciet gemaakt worden, zodat alle procedure-oproepen en hoofdingen van de vorm $p(X_1, \ldots, X_n)$ zijn, waarbij de X_i verschillende variabelen zijn. De analyse van niet-genormaliseerde programma's komt later aan bod.

3.3 Primitieve abstracte operaties

3.3.1 Abstracte conjunctie

Beschouw $AC_1, AC_2 \in Con^a$. Abstracte conjunctie $AC_1 \wedge AC_2$ is de tegenhanger van de concrete conjunctie van twee verzamelingen constraints (deze laatste is gedefinieerd als $CS_1 \wedge CS_2 = \{C_1 \wedge C_2 \mid C_1 \in CS_1, C_2 \in CS_2\}$). Het domein van $CS_1 \wedge CS_2$ (resp. $AC_1 \wedge AC_2$) is de unie van de domeinen van CS_1 en CS_2 (resp. AC_1 en AC_2).

De voorwaarde voor correctheid is :
als $\alpha(CS_1) \leq^a AC_1$ en $\alpha(CS_2) \leq^a AC_2$, dan moet $\alpha(CS_1 \wedge CS_2) \leq^a AC_1 \wedge AC_2$.

3.3.2 Abstracte projectie

Beschouw $AC \in Con^a$. De abstracte projectie van AC op een verzameling variabelen V, genoteerd $\exists_V AC$, beperkt de aandacht tot die variabelen en bootst de concrete projectie $\exists_V CS$ na, met $\exists_V CS = \{\exists_V C \mid C \in CS\}$[4]. Het domein van $\exists_V CS$ en $\exists_V AC$ is V.

De voorwaarde voor correctheid is :
als $\alpha(CS) \leq^a AC$, dan moet $\alpha(\exists_V CS) \leq^a \exists_V AC$.

[3] D.i. als $(Con^c_D, \leq^c) \overset{\gamma}{\underset{\alpha}{\rightleftarrows}} (Con^a_D, \leq^a)$ een Galois connectie [2] is, wat betekent dat $(Con^c_D, \leq^c)$ en $(Con^a_D, \leq^a)$ partieel geordende structuren zijn, α en γ monotoon zijn en $\forall AC \in Con^a_D : \alpha(\gamma(AC)) \leq^a AC$ & $\forall CS \in Con^c_D : CS \leq^c \gamma(\alpha(CS))$.

[4] $\exists_V C = \exists X_1, \ldots, \exists X_n . C$ waarbij $\{X_1, \ldots, X_n\} = vars(C) \setminus V$ en waarbij $vars(s)$ de verzameling is van variabelen in het syntactisch object s.

3.4 Andere abstracte operaties

3.4.1 Abstracte interpretatie van een constraint

Deze operatie is de tegenhanger van het toevoegen van een constraint C aan een verzameling constraints CS (verzameld tot op het huidig punt van uitvoering). De abstracte terugkeer-constraint AC_s van C wordt gedefinieerd als de abstracte conjunctie van de abstracte oproep-constraint AC_c van C en de abstractie van C : $AC_s = AC_c \wedge \alpha(C)$.

De correctheid volgt uit de correctheid van de abstracte conjunctie.

3.4.2 Procedure-oproep$(A, AC_c) = \{AC^1_{in}, \ldots, AC^m_{in}\}$

Voor elke programma-regel Cl^j die een definitie is voor de oproep A, berekent deze operatie de abstracte oproep-constraint AC^j_{in} van de eerste constraint of oproep in Cl^j. Hierbij wordt de oproep-constraint AC_c van A geprojecteerd op de variabelen van de oproep A, d.i. $AC_{entry} = \overline{\exists}_{vars(A)} AC_c$; daarna wordt deze AC_{entry} hernoemd naar de variabelen in Cl^j en eventueel uitgebreid met informatie over de lokale variabelen in Cl^j (het domein van AC^j_{in} is de verzameling van variabelen in Cl^j). Dit geeft het resultaat AC^j_{in}.

De voorwaarden voor correctheid zijn (met ρ de hernoemingsfunctie die de variabelen van de oproep A hernoemt naar de variabelen in de hoofding van Cl^j) :

1. als $\alpha(CS_c) \leq^a AC_c$, dan moet $\alpha(\overline{\exists}_{vars(A)} CS_c) \leq^a AC_{entry}$ (d.i. correctheid van abstracte projectie);
2. als $\alpha(\overline{\exists}_{vars(A)} CS_c) \leq^a AC_{entry}$, dan moet $\alpha((\overline{\exists}_{vars(A)} CS_c)\rho) \leq^a AC^j_{in}$.

3.4.3 Procedure-terugkeer$(A, AC_c, \{H^1, \ldots, H^m\}, \{AC^1_{out}, \ldots, AC^m_{out}\}) = AC_s$

Als in elke programma-regel $Cl^j : H^j \leftarrow B^j$ die de oproep A definieert de terugkeer-constraint van de laatste constraint of oproep bekend is, dan berekenen we eerst AC_{exit}. Dit is de bovengrens van de AC^j_{out} die geprojecteerd werden op $vars(H^j)$ en hernoemd naar $vars(A)$. Daarna wordt de terugkeer-constraint AC_s van A berekend door combinatie van AC_c en AC_{exit}. Een mogelijke definitie voor AC_s is : $AC_s = AC_c \wedge AC_{exit}$. In sommige gevallen kan een hogere precisie verkregen worden door AC_s niet direct via abstracte conjunctie te definiëren. Dit is het geval voor de analyses die zullen voorgesteld worden in de volgende hoofdstukken. Abstracte conjunctie kan dan echter nog altijd toegepast worden op componenten van de abstracte constraints.

De voorwaarden voor correctheid zijn :

1. als $\alpha(CS^j_{out}) \leq^a AC^j_{out}$, dan moet $\alpha((\overline{\exists}_{vars(H^j)} CS^j_{out})\rho^{-1}) \leq^a AC_{exit}$;
2. als $\alpha(CS_c) \leq^a AC_c$ en $\alpha(CS_{exit}) \leq^a AC_{exit}$, waarbij er een $CS_{local} \in Con^c$ over $vars(A)$ bestaat zodat $CS_{exit} = (\overline{\exists}_{vars(A)} CS_c) \wedge CS_{local}$, dan moet $\alpha(CS_c \wedge CS_{exit}) \leq^a AC_s$.

De correctheid steunt op de correctheid van abstracte conjunctie en projectie en van de bovengrens operatie.

4 Analyse van vrijheidsgraden

Binnen het aangepast raamwerk wordt een analyse ontworpen die informatie afleidt (1) over variabelen die optreden als *vrijheidsgraden* in constraints en (2) over mogelijke afhan-

kelijkheden tussen variabelen.

Een variabele X treedt op als vrijheidsgraad, of kortweg is *vrij*, in een constraint C als X nog alle mogelijke waarden toegelaten door zijn type kan aannemen. Toegespitst op CLP(H,N) wordt de definitie :

Definitie 4.1 (Vrije variabele)
Een variabele X is vrij in een constraint C als (1) C geen constraint $X \diamond t$ impliceert waarbij t een Herbrand term is verschillend van een variabele en $\diamond \in \{=, \neq\}$, en als (2) C geen constraint $X \diamond n$ impliceert waarbij n een getal is en $\diamond \in \{=, \neq, >, \geq\}$.

We zeggen ook dat C een variabele X *beperkt* als X niet-vrij is in C.

De cruciale operatie bij het detecteren van vrije variabelen is constraint-implicatie. Geïmpliceerde constraints die het niet-vrij zijn van variabelen bepalen komen aan het licht door een constraint te transformeren naar opgeloste vorm.

Voorbeeld 4.1
De opgeloste vorm van $X + Y = 6 \wedge X + Y - Z = 2$ is $X + Y = 6 \wedge Z = 4$; dus X en Y zijn vrij (hoewel ze beperkt worden tot het numerieke type) en Z is niet-vrij.

De constraint $X = f(A) \wedge X = Y \wedge B + A = 0$ heeft als opgeloste vorm $X = f(A) \wedge Y = f(A) \wedge B + A = 0$. De variabelen X en Y zijn niet-vrij, A en B zijn vrij.

Informatie over afhankelijkheden tussen variabelen is nodig om nauwkeurige informatie over vrijheidsgraden te verkrijgen. Bijvoorbeeld, veronderstel dat een variabele X beperkt wordt via $X > 5$; een andere vrije variabele Y blijft vrij indien geen afhankelijkheid tussen X en Y bestaat, anders wordt Y ook niet-vrij. Afhankelijkheden worden teweeggebracht door constraints tussen de variabelen. In tegenstelling tot het LP geval leiden niet enkel unificatie constraints tot afhankelijkheden, maar ook numerieke constraints of een combinatie van beide. Vooral dit laatste verhoogt de complexiteit. Constraint-implicatie is opnieuw de basisoperatie bij het bepalen van de afhankelijkheden.

Definitie 4.2 (Strikte afhankelijkheid)
De constraint C creëert een strikte afhankelijkheid tussen een verzameling van variabelen $\{X_1, \ldots, X_n\}$ $(n \geq 2)$ indien $\exists X_j (1 \leq j \leq n)$ zodat $[\ \forall i (1 \leq i \leq n, i \neq j) : \exists v_i$ toegelaten door het type van X_i zodat $C \wedge x_1 = v_1 \wedge \ldots \wedge x_{j-1} = v_{j-1} \wedge x_{j+1} = v_{j+1} \wedge \ldots \wedge x_n = v_n$ de variabele X_j beperkt, en er is geen deelverzameling van de X_i die X_j beperkt $]$.

Intuïtief betekent deze definitie : de constraint C creëert een strikte afhankelijkheid tussen een verzameling variabelen $\{X_1, \ldots, X_n\}$ $(n \geq 2)$ indien (verder) beperken van alle variabelen behalve één de overblijvende variabele (verder) kan beperken.

Definitie 4.3 (Afhankelijkheid)
De constraint C creëert een afhankelijkheid tussen een verzameling variabelen $\{X_1, \ldots, X_n\}$ $(n \geq 2)$ indien C een strikte afhankelijkheid tussen $\{X_1, \ldots, X_n\}$ teweegbrengt of indien C verschillende strikte afhankelijkheden $S_1, \ldots, S_m$ $(m \geq 2)$ creëert waarbij $S_1 \cup \ldots \cup S_m = \{X_1, \ldots, X_n\}$.

Een afhankelijkheid geeft aan dat het niet-vrij zijn van variabelen *kan* gepropageerd worden, i.e. propagatie is mogelijk maar niet zeker. Dit wordt geïllustreerd in onderstaande voorbeelden.

Voorbeeld 4.2
Beschouw de constraint $X = f(A) \wedge A + B = 3$. *De primitieve unificatie constraint leidt tot de (strikte) afhankelijkheid* $\{X, A\}$. *De numerieke gelijkheid creëert de (strikte) afhankelijkheid* $\{A, B\}$. *Combinatie van de twee leidt tot* $\{X, B\}$. *Als* X *verder beperkt wordt, dan wordt* A *en daardoor ook* B *verder beperkt; omgekeerd, verder beperken van* B *zal* X *verder beperken.*

Beschouw de constraint $X = f(A, B) \wedge A + B - T = 2$. *Verder beperken van* X kan *eventueel tegelijk* A *en* B *beperken, waardoor ook* T *beperkt wordt. Omgekeerd, beperken van* T *beperkt* X *verder in de zin dat de som van zijn componenten beperkt wordt (b.v. toevoegen van* $T = 1$ *betekent dat* $X = f(4, 5)$ *niet meer kan, dus* X *is beperkt). Dus de constraint creëert de (strikte) afhankelijkheid* $\{X, T\}$ *(naast de strikte afhankelijkheden* $\{X, A\}$, $\{X, B\}$ *en* $\{A, B, T\}$*).*

Aangezien de eigenschap "vrij zijn" van een variabele in het algemeen onbeslisbaar is, kan op abstract niveau slechts een benadering berekend worden. Dit leidt tot de noties *zeker vrij zijn* en *mogelijk niet-vrij zijn* van een variabele. Om correcte propagatie van mogelijk niet-vrij zijn te garanderen, moeten alle *mogelijke* afhankelijkheden beschouwd worden.

4.1 Concreet en abstract domein

Een element van het concreet domein $Con^{\mathcal{C}}$ is een verzameling van constraints. Een element van het abstract domein $Con^{\mathcal{F}}$, d.i. een abstracte constraint, is ofwel een verzameling van verzamelingen van variabelen ofwel het speciaal minimaal element $\perp$. De informatie over het vrij zijn van variabelen wordt geïntegreerd met de informatie over afhankelijkheden tussen variabelen : singletons geven aan welke variabelen mogelijk niet-vrij zijn, niet-singletons stellen mogelijke afhankelijkheden voor.

Definitie 4.4 (Abstract domein)
Beschouw de verzameling van programma-variabelen Var.
Dan $Con^{\mathcal{F}} = \wp(\wp_{\emptyset}(Var)) \cup \{\perp\}$ *met* $\wp_{\emptyset}(S) = \wp(S) \setminus \{\emptyset\}$ *en* $\wp(S)$ *is de delenverzameling van* S.

Definitie 4.5 (Abstracte orde relatie)

$$\forall AC_1, AC_2 \in Con^{\mathcal{F}} : AC_1 \leq^{\mathcal{F}} AC_2 \text{ asa } AC_1 = \perp \text{ of } (AC_1 \neq \perp, AC_2 \neq \perp, \text{ en } AC_1 \subseteq AC_2).$$

Het maximaal element in $Con^{\mathcal{F}}$ is $\wp_{\emptyset}(Var)$, het minimaal element is $\perp$.

Definitie 4.6 (Kleinste bovengrens operatie ("least upper bound"))

$$\forall AC_1, AC_2 \in Con^{\mathcal{F}} : lub^{\mathcal{F}}(AC_1, AC_2) = \begin{cases} AC_1 & \text{als } AC_2 = \perp, \\ AC_2 & \text{als } AC_1 = \perp, \\ AC_1 \cup AC_2 & \text{anders.} \end{cases}$$

Deze operatie kan gemakkelijk veralgemeend worden tot het bepalen van de kleinste bovengrens van een *verzameling* van abstracte constraints.

De verwijdingsoperatie ("widening") W kan gelijkgesteld worden aan $lub^{\mathcal{F}}$ vermits $Con^{\mathcal{F}}$ eindig is.

De bovenstaande definities zorgen ervoor dat $Con^{\mathcal{F}}$ de gevraagde algebraïsche structuur heeft. Meer zelfs, $(Con^{\mathcal{C}}, \subseteq) \underset{\gamma}{\overset{\alpha}{\rightleftarrows}} (Con^{\mathcal{F}}, \leq^{\mathcal{F}})$ is een Galois connectie.

Definitie 4.7 (Abstractie van een verzameling constraints)

$\forall CS \in Con^{\mathcal{C}} : \alpha(CS) = \bot$ *als* $CS = \emptyset$,
$= lub^{\mathcal{F}}(\{\ \alpha(\{C\})\ \mid\ C \in CS\ \})$ *anders.*

In het vervolg wordt $\alpha(\{C\})$ genoteerd als $\alpha(C)$.

Definitie 4.8 (Abstractie van een constraint)

$\alpha(C) = \bot$ *als* $sform(C) = false$ *(d.i. C is onoplosbaar),*
$= \{\{X\}\ \mid\ X \in vars(C), C$ *beperkt* $X\} \cup$
$\{\{X_1, \ldots, X_n\}\ \mid\ C$ *creëert de afhankelijkheid* $\{X_1, \ldots, X_n\} \subseteq vars(C)\}$ *anders.*

Dus voor C met $sform(C) = true$ is de abstractie $\alpha(C) = \emptyset$.

Deze algemene definitie wordt nu nauwkeuriger gespecificeerd voor een (gemengde) constraint C in $CLP(H, N)$. Hierbij wordt de opgeloste vorm van C gebruikt.
Voor *numerieke* constraints gaat de aandacht eerst naar lineaire gelijkheden. Elke numerieke gelijkheid creëert een afhankelijkheid tussen zijn variabelen. Als de gelijkheid slechts één variabele bevat, dan is die niet-vrij. Om alle niet-vrije variabelen en afhankelijkheden te bepalen, moeten alle mogelijke lineaire combinaties van de gelijkheden beschouwd worden. Deze worden voortgebracht door de primitieve constraints in een opgeloste vorm van de set van gelijkheden. Merk op dat de abstractie enkel de variabelen uit een gelijkheid bijhoudt en geen informatie bevat over de coëfficiënten van de variabelen.
Elke primitieve *unificatie* constraint van de vorm $X = t$ in $sform(unif^*(C))$ creëert een afhankelijkheid tussen X en elke variabele in t. Als t geen variabele is, dan is X niet-vrij wat voorgesteld wordt door $\{X\}$ in de abstractie.
Een *gemengde* afhankelijkheid onstaat door de combinatie van een primitieve constraint in $sform(num^*(C))$ met één of meer primitieve unificatie constraints in $sform(unif^*(C))$. Laat $T_i[Y_i]$ een Herbrand term voorstellen die Y_i bevat; als $X_i = T_i[Y_i]$ $(1 \leq i \leq m)$ behoort tot $sform(unif^*(C))$ en $a_1 Y_1 + \ldots + a_m Y_m + a'_1 Z_1 + \ldots + a'_n Z_n = b$ behoort tot $sform(num^*(C))$, dan ontstaat een afhankelijkheid $\{X_1, \ldots, X_m, Z_1, \ldots, Z_n\}$. Merk op dat bij $m > 1$ verschillende Y_i *tegelijkertijd* kunnen beperkt worden door de overeenkomstige X_i; de X_i zijn niet noodzakelijk verschillend, want één X_i kan tegelijk verschillende van zijn Y componenten beperken.

Definitie 4.9 (Abstractie van een constraint in CLP(H,N))
C is een (oplosbare) constraint, θ is de substitutie die overeenkomt met $sform(unif^*(C))$,

$$W = \left\{ \{X\} \,\middle|\, X\theta \text{ is geen variabele} \right\} \cup \left\{ \{X, Y\} \,\middle|\, X \not\equiv Y,\ vars(X\theta) \cap vars(Y\theta) \neq \emptyset \right\}$$

$$\cup \left\{ \{X_1, \ldots, X_n\} \;\middle|\; \begin{array}{l}(a_1X_1 + \ldots + a_nX_n = b) \in sform(num^*(C)) \\ voor \text{ om het even welke } sform(num^*(C))\end{array} \right\}^5$$

$$\cup \left\{ \begin{array}{l} \{X_1, \ldots, X_m, Z_1, \ldots, Z_n\} \mid \\ Y_i \in vars(X_i\theta),\ alle\ Y_i\ zijn\ verschillend, \\ de\ X_i\ zijn\ niet\ noodzakelijk\ verschillend\ (1 \leq i \leq m), \\ (a_1Y_1 + \ldots + a_mY_m + a_1'Z_1 + \ldots + a_n'Z_n = b) \in sform(num^*(C)) \end{array} \right\}$$

Dan $\alpha(C) = close(W)$ *waarbij* $close(W)$ *de sluiting onder unie van* W *berekent.*

Voorbeeld 4.3
Beschouw de constraint $X = f(A) \wedge U = g(B) \wedge T + A - 3B = 1$. *Dan*

$$sform(unif^*(C)) \equiv X = f(A) \wedge U = g(B),$$
$$sform(num^*(C)) \equiv T + A - 3B = 1,$$

$$W = \left\{ \begin{array}{l} \{X\}, \{X, A\}, \{U\}, \{U, B\}, \{A, B, T\}, \\ \{X, B, T\}, \{A, U, T\}, \{X, U, T\} \end{array} \right\}$$

$$\alpha(C) = close(W)$$
$$= \left\{ \begin{array}{l} \{X\}, \{X, A\}, \{U\}, \{U, B\}, \{A, B, T\}, \{X, B, T\}, \\ \{A, U, T\}, \{X, U, T\}, \{X, A\}, \{X, U\}, \{X, A, U\}, \{X, B, U\}, \\ \{X, A, B, U\}, \{X, A, B, T\}, \{A, B, U, T\}, \{X, B, U, T\}, \\ \{X, A, U, T\}, \{X, A, B, U, T\} \end{array} \right\}$$

Verschillen ($\neq$) en ongelijkheden ($>, \geq$) worden in eerste instantie geabstraheerd zoals gelijkheden; dit is een ruwe abstractie. Ook voor een niet-lineaire constraint C wordt een ruwe abstractie genomen : $\alpha(C) = \wp_\emptyset(vars(C))$. In sectie 6 wordt aangehaald hoe de precisie kan verbeterd worden.
Het nemen van de sluiting in Definitie 4.9, wat neerkomt op het exhaustief voorstellen van alle afhankelijkheden, verhoogt de precisie van de verdere analyse. Het sluiten zou uitgesteld kunnen worden tot op het moment van abstracte conjunctie, waar combinaties (unies) van afhankelijkheden beschouwd moeten worden. Dit zou echter inhouden dat ook afhankelijkheden die afkomstig zijn van verschillende uitvoeringspaden (OF-takken in de EN-OF boom) en dus niets met elkaar te maken hebben, toch gecombineerd zouden worden. Anderzijds zou het niet nemen van de sluiting in Definitie 4.9 de grootte van de abstracties fel beperken. De minimale analyse, die verder besproken wordt, ruilt juist op deze manier precisie in voor efficiëntie.

4.2 Primitieve abstracte operaties

Definitie 4.10 (Abstracte conjunctie)
De abstracte conjunctie van $AC_1, AC_2 \in Con^{\mathcal{F}}$ *wordt genoteerd als* $AC_1 \wedge AC_2$.
1. *Als* $AC_1 = \bot$ *en/of* $AC_2 = \bot$, *dan* $AC_1 \wedge AC_2 = \bot$;

[5] Elke opgeloste vorm, overeenkomend met een bepaalde parameterkeuze, maakt andere afhankelijkheden expliciet.

2. anders

$$AC_1 \wedge AC_2 = AC_1 \cup AC_2 \cup (AC_1 \oplus AC_2) \quad \textit{waarbij}$$
$$AC_1 \oplus AC_2 = \left\{ (A_1 \cup A_2) \setminus D \;\middle|\; A_1 \in AC_1, A_2 \in AC_2, D \subseteq A_1 \cap A_2 \right\} \setminus \{\emptyset\}.$$

Het $AC_1 \oplus AC_2$ gedeelte berekent de mogelijke afhankelijkheden die onstaan door afhankelijkheden in AC_1 te combineren met die in AC_2.

Theorema 4.1 (Correctheid van abstracte conjunctie)
Als $\alpha(CS_1) \leq^{\mathcal{F}} AC_1$ *en* $\alpha(CS_2) \leq^{\mathcal{F}} AC_2$, *dan* $\alpha(CS_1 \wedge CS_2) \leq^{\mathcal{F}} AC_1 \wedge AC_2$.

Definitie 4.11 (Abstracte projectie)
De abstracte projectie van $AC \in Con^{\mathcal{F}}$ *op* $V \subseteq Var$ *wordt genoteerd als* $\exists_V AC$.

1. *Als* $AC = \bot$, *dan* $\exists_V AC = \bot$;
2. *anders* $\exists_V AC = \{S \in AC \mid S \subseteq V\}$.

Theorema 4.2 (Correctheid van abstracte projectie)
Als $\alpha(CS) \leq^{\mathcal{F}} AC$, *dan* $\alpha(\exists_V CS) \leq^{\mathcal{F}} \exists_V AC$.

4.3 Andere abstracte operaties

Om de definities van de operaties te vereenvoudigen wordt het $\bot$ geval niet expliciet aangegeven. Zodra op een programmapunt $\bot$ wordt bekomen, wordt dit verder doorgegeven naar de volgende programmapunten tot op het einde van een procedure-definitie. Als alle definities $\bot$ opleveren, wordt $\bot$ ook doorgegeven naar de omgeving van de procedure-oproep.

4.3.1 Samengestelde abstracte constraint

Vooraleer de andere abstracte operaties te definiëren, bespreken we eerst een aspect in verband met de precisie van de analyse. Een abstracte constraint wordt opgesplitst in twee componenten om onderscheid te kunnen maken tussen *oude* informatie die doorgegeven wordt bij een procedure-oproep, en *nieuwe* informatie, die verzameld wordt tijdens de lokale analyse van een procedure. Bij procedure-terugkeer is het essentieel om de oude informatie niet te combineren met de informatie in de oproep-constraint van de procedure. Tweemaal toevoegen van eenzelfde numerieke constraint aan een verzameling constraints leidt namelijk tot het mogelijk niet-vrij zijn van de betrokken variabelen. De reden is dat de abstractie de coefficiënten van variabelen niet bijhoudt. Beschouw een constraint $a_1X_1 + \ldots + a_nX_n = a_{n+1}$ met als abstractie $\{\{X_1, \ldots, X_n\}\}$. Een tweede maal toevoegen van deze constraint (d.w.z. de abstracte conjunctie nemen van $\{\{X_1, \ldots, X_n\}\}$ en een abstracte constraint AC die $\{X_1, \ldots, X_n\}$ al bevat) leidt tot de aanwezigheid van $\{X_i\}$ $(1 \leq i \leq n)$ in de resulterende abstracte constraint AC'. Immers, de verzameling $\{X_1, \ldots, X_n\}$ in AC stelt een vergelijking voor van de vorm $b_1X_1 + \ldots + b_nX_n = b_{n+1}$ waarbij de coefficiënten b_j niet langer gekend zijn. Combinatie van zo'n constraint met $a_1X_1 + \ldots + a_nX_n = a_{n+1}$ kan, mits een goede keuze van de b_j, leiden tot een lineaire combinatie die enkel X_i bevat. De preciese definitie van een samengestelde abstracte constraint luidt als volgt :

Definitie 4.12 (Samengestelde abstracte constraint)
Een samengestelde abstracte constraint AC *is ofwel* $\bot$ *ofwel een paar* (AC^o, AC^n) *waarbij*

$AC^o, AC^n \in \wp(\wp_\emptyset(Var))$. *$AC^o$ bevat de informatie doorgegeven bij procedure-oproep en de combinatie daarvan met de lokale informatie uit de procedure; AC^n bevat de lokale informatie verzameld tijdens de analyse van de procedure.*

Definitie 4.13 (AC^t)
Beschouw een samengestelde abstracte constraint (AC^o, AC^n). Dan $AC^t = AC^o \cup AC^n$.

4.3.2 Abstracte interpretatie van een constraint

De abstracte interpretatie van een constraint C berekent de terugkeer-constraint AC_s van C door $\alpha(C)$ toe te voegen aan de oproep-constraint AC_c. Hierbij wordt $\alpha(C)$ zelf, en zijn combinatie met de lokale informatie, in AC_s^n geplaatst, terwijl de combinatie met oude informatie (doorgegeven bij procedure-oproep) in AC_s^o geplaatst wordt.

Definitie 4.14 (Abstracte interpretatie van een constraint)
De abstracte terugkeer-constraint (AC_s^o, AC_s^n) van C is gedefinieerd als :

1. $AC_s^o = AC_c^o \cup (AC_c^o \oplus \alpha(C))$ *en*
2. $AC_s^n = AC_c^n \wedge \alpha(C)$.

De correctheid van de operatie steunt op het feit dat $AC_s^t = AC_c^t \wedge \alpha(C)$ en op de correctheid van abstracte conjunctie.

4.3.3 Procedure-oproep

Veronderstel dat de oproep $p(Y_1, \ldots, Y_k)$ is en dat $\{X_1, \ldots, X_m, Y_1, \ldots, Y_k\}$ het domein is van zijn oproep-constraint (AC_c^o, AC_c^n). De procedure-oproep operatie berekent voor elke definitie $p(Z_1, \ldots, Z_k) \leftarrow B_1, \ldots, B_l$ van p/k de oproep-constraint van de eerste constraint of oproep B_1. Hierbij wordt AC_c^t geprojecteerd op $\{Y_1, \ldots, Y_k\}$ en hernoemd naar $\{Z_1, \ldots, Z_k\}$. Dit levert de o-component op van de oproep-constraint van B_1. De n-component krijgt als initiële waarde $\emptyset$.

Definitie 4.15 (Procedure-oproep)
De abstracte oproep-constraint (AC_{in}^o, AC_{in}^n) van B_1 is gedefinieerd als $(AC_{in}^o, AC_{in}^n) = (AC_{entry}^o, AC_{entry}^n)\rho$ waarbij $AC_{entry} = (\overline{\exists}_{\{Y_1,\ldots,Y_k\}} AC_c^t, \emptyset)$ en $\rho = \{Y_1 \leftarrow Z_1, \ldots, Y_k \leftarrow Z_k\}$ (hernoeming).

De correctheid van procedure-oproep steunt op het feit dat $AC_{in}^t = (\overline{\exists}_{\{Y_1,\ldots,Y_k\}} AC_c^t)\rho$ en op de correctheid van abstracte projectie.

4.3.4 Procedure-terugkeer

Veronderstel dat de oproep $p(Y_1, \ldots, Y_k)$ is en dat $\{X_1, \ldots, X_m, Y_1, \ldots, Y_k\}$ het domein is van zijn oproep-constraint (AC_c^o, AC_c^n). Veronderstel ook dat (AC_i^o, AC_i^n) de abstracte constraint is op het einde van de i^{de} definitie voor p/k en dat de hoofding van elke definitie van de vorm $p(Z_1, \ldots, Z_k)$ is. Eerst wordt AC_{exit}^n berekend, die de *lokale* informatie uit de verschillende definities samenbundelt (via de bovengrens operatie). Daarna worden AC_c en AC_{exit}^n gecombineerd tot de terugkeer-constraint AC_s. Zoals vroeger aangegeven is het belangrijk om de informatie die doorgegeven werd bij procedure-oproep, niet nogmaals te combineren met AC_c. Anders wordt het resultaat te onnauwkeurig.

Definitie 4.16 (Procedure-terugkeer)
$AC^n_{exit} = lub^{\mathcal{F}}(\{\ldots,(\overline{\exists}_{\{z_1,\ldots,z_k\}} AC^n_i)\rho^{-1},\ldots\})$ *waarbij* $\rho^{-1} = \{Z_1 \leftarrow Y_1,\ldots,Z_k \leftarrow Y_k\}$ *(hernoeming).*
De abstracte terugkeer-constraint (AC^o_s, AC^n_s) *van de oproep* $p(Y_1,\ldots,Y_k)$ *is gedefinieerd als :*

1. $AC^o_s = AC^o_c \cup (AC^o_c \oplus AC^n_{exit})$
2. $AC^n_s = AC^n_c \wedge AC^n_{exit}$.

De correctheid van procedure-terugkeer steunt op het feit dat $AC^t_s = AC^t_c \wedge AC^n_{exit}$ en op de correctheid van abstracte conjunctie, abstracte projectie en $lub^{\mathcal{F}}$.

4.4 Voorbeeld

Beschouw het volgend eenvoudig programma, dat zowel numerieke als unificatie constraints bevat.

?- (AC_0) $\mathbf{Z = 1}$, (AC_1) $\mathbf{p(X,Y,Z,T)}$ (AC_2).

$\mathbf{p(X,Y,Z,T)} \leftarrow (AC_3)$ $\mathbf{X = f(Z)}$, (AC_4) $\mathbf{Z + T = 0}$ (AC_5).
$\mathbf{p(X,Y,Z,T)} \leftarrow (AC_6)$ $\mathbf{X = g(Y)}$, (AC_7) $\mathbf{Y - T = 0}$ (AC_8).

Veronderstel dat het programma gestart wordt met een lege verzameling van constraints. De samengestelde abstracte constraints op elk programmapunt zijn dan :

$AC_0 = (\emptyset,\ \emptyset)$
$AC_1 = (\emptyset,\ \{\{Z\}\})$
$AC_2 = (\emptyset,\ \{\{Z\},\{X\},\{X,Z\},\{Z,T\},\{T\},\{X,T\},\{X,Z,T\},\{X,Y\},\{X,Y,Z\},$
$\{Y,T\},\{Y,Z,T\},\{X,Y,T\},\{X,Y,Z,T\}\})$

$AC_3 = (\{\{Z\}\},\ \emptyset)$
$AC_4 = (\{\{Z\},\{X,Z\},\{X\}\},\ \{\{X\},\{X,Z\}\})$
$AC_5 = (\{\{Z\},\{X,Z\},\{X\},\{T\},\{X,T\},\{X,Z,T\}\},\ \{\{X\},\{X,Z\},\{Z,T\},\{X,Z,T\},$
$\{X,T\}\})$

$AC_6 = (\{\{Z\}\},\emptyset)$
$AC_7 = (\{\{Z\},\{X,Z\},\{X,Y,Z\}\},\ \{\{X\},\{X,Y\}\})$
$AC_8 = (\{\{Z\},\{X,Z\},\{X,Y,Z\},\{Y,Z,T\},\{X,Y,Z,T\},\{X,Z,T\}\},\ \{\{X\},\{X,Y\},$
$\{Y,T\},\{X,Y,T\},\{X,T\}\})$

In AC_3 en AC_6 gaat door procedure-oproep $\{Z\}$ over van de n-component naar de o-component; $\{Z\}$ is informatie die vanuit de oproep-omgeving wordt doorgegeven naar de procedure. AC_5 geeft aan dat op het einde van de eerste definitie voor $p/4$ de variabelen X, Z en T mogelijk niet-vrij zijn (de singletons $\{X\}$, $\{Z\}$ en $\{T\}$ behoren tot AC_5) en Y zeker vrij is. Dit komt precies overeen met het concreet geval. Ook op het einde van de tweede definitie wordt nauwkeurige informatie afgeleid : enkel X en Z zijn mogelijk niet-vrij, Y en T zijn zeker vrij. In AC_2 worden bij procedure-terugkeer de resultaten van de twee definities samengebracht, vandaar dat X, Z en T mogelijk niet-vrij zijn en Y zeker vrij is.

5 Optimalisaties

De analyse van vrijheidsgraden zoals ze werd voorgesteld in de vorige sectie is niet erg efficiënt, zowel wat het tijds- als geheugengebruik betreft. Abstracte constraints kunnen vrij groot worden, vermits alle afhankelijkheden exhaustief opgesomd worden. De analyse vormt wel de theoretische basis voor de meer praktische varianten in deze sectie. We beschrijven twee methodes om een compactere voorstelling van abstracte constraints te bekomen.

5.1 Minimale analyse

De minimale analyse bestaat erin enkel *minimale* afhankelijkheden bij te houden. Dit zijn afhankelijkheden die niet via combinatie (unie) van andere afhankelijkheden kunnen bekomen worden. Beschouw bijvoorbeeld de constraint $C \equiv X + Y = 3 \wedge 2Y + Z = 0$ met $\alpha^{\mathcal{F}}(C) = \{\{X, Y\}, \{Y, Z\}, \{X, Z\}, \{X, Y, Z\}\}$; C impliceert b.v. $X + 3Y + Z = 3$ die de afhankelijkheid $\{X, Y, Z\}$ veroorzaakt. Deze afhankelijkheid kan echter ook verkregen worden via unie van b.v. $\{X, Y\}$ en $\{Y, Z\}$.

Definitie 5.1 (Minimale verzameling)
Beschouw $SS \in \wp(\wp_\emptyset(Var))$. *Een* $S \in SS$ *is minimaal in* SS *asa* $\nexists S_1, \ldots, S_m \in SS \setminus \{S\}$ $(m \geq 2)$ *zodat* $S = S_1 \cup \ldots \cup S_m$.

Definitie 5.2 (min)
Beschouw $AC \in Con^{\mathcal{F}}$. *Dan*

$$min(AC) = \begin{cases} \bot & \text{als } AC = \bot \\ \{S \in AC \mid S \text{ is een minimale verzameling in } AC\} & \text{anders.} \end{cases}$$

Definitie 5.3 (Minimale abstracte constraint)
Beschouw $AC \in \wp(\wp_\emptyset(Var))$. AC *is minimaal asa* $min(AC) = AC$.

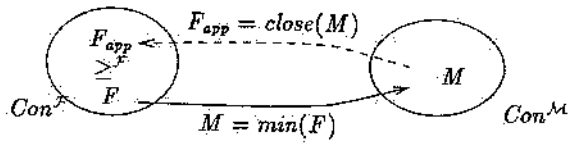
De minimale abstractie van een verzameling constraints kan berekend worden via minimalisatie van de originele abstractie. Het superscript $\mathcal{M}$ verwijst naar de minimale analyse, $\mathcal{F}$ verwijst naar de originele analyse.

Definitie 5.4 (Minimale abstractie van een verzameling constraints)
Beschouw $CS \in Con^{\mathcal{C}}$. *Dan* $\alpha^{\mathcal{M}}(CS) = min(\alpha^{\mathcal{F}}(CS))$.

$\alpha^{\mathcal{F}}(CS)$ werd gedefinieerd in functie van $\alpha^{\mathcal{F}}(C)$ met $C \in CS$ (zie Definitie 4.7). Een meer directe definitie van $\alpha^{\mathcal{M}}(C)$ is mogelijk : de operatie $close(W)$ in Definitie 4.9 voegt slechts niet-minimale afhankelijkheden toe, dus alleen W hoeft berekend te worden.

Voorbeeld 5.1
Beschouw de constraint $X = f(Y) \wedge Y = g(U) \wedge U - T - Z = 0$. *Dan,*
$W = \{\{X\}, \{Y\}, \{X, Y\}, \{X, U\}, \{Y, U\}, \{U, T, Z\}, \{X, T, Z\}, \{Y, T, Z\}\}$,
$\alpha^{\mathcal{F}}(C) = close(W)$,
$\alpha^{\mathcal{M}}(C) = min(W) = \{\{X\}, \{Y\}, \{X, U\}, \{Y, U\}, \{U, T, Z\}, \{X, T, Z\}, \{Y, T, Z\}\}$.
De minimale abstractie is heel wat compacter dan de originele abstractie.

Het verband tussen de minimale en originele abstractie van $CS \in Con^{\mathcal{C}}$ wordt geïllustreerd in Figuur 1, waarbij $F = \alpha^{\mathcal{F}}(CS)$ en $M = \alpha^{\mathcal{M}}(CS)$. Omdat niet alle abstracte constraints gesloten zijn onder unie, kan het minimalisatieproces leiden tot een verlies van precisie/expressiviteit (hoewel dit in de praktijk zelden voorkomt). In het algemeen geldt dus $F \leq^{\mathcal{F}} close(M)$.



Figuur 1: Verband tussen de $\mathcal{F}$ en $\mathcal{M}$ abstractie

Minimale abstracte constraints laten toe heel wat efficiëntere (primitieve) abstracte operaties te definiëren. Bij abstracte conjunctie wordt slechts een gedeeltelijke sluiting van de abstracte constraints berekend. Daarnaast wordt het genereren van niet-minimale afhankelijkheden zoveel mogelijk vermeden.

5.2 Gebruik van "definiteness" informatie

Een tweede methode om de grootte van abstracte constraints te beperken is het benutten van "definiteness" informatie. Deze informatie geeft aan welke variabelen beperkt zijn tot een unieke waarde. De originele abstractie van een verzameling constraints CS, d.i. $\alpha^{\mathcal{F}}(CS)$, kan gesplitst worden in een deel dat geen uniek beperkte variabelen bevat – $compl(D, \alpha^{\mathcal{F}}(CS))$ – en een deel dat wel uniek beperkte variabelen bevat – $defrelated(D, \alpha^{\mathcal{F}}(CS))$.

Stelling 5.1
Veronderstel dat D de verzameling is van uniek beperkte variabelen in CS of een deelverzameling daarvan. Dan geldt : $\alpha^{\mathcal{F}}(CS) = compl(D, \alpha^{\mathcal{F}}(CS)) \cup defrelated(D, \alpha^{\mathcal{F}}(CS))$ waarbij

- $compl(D, SS) = \{S \in SS \mid S \cap D = \emptyset\}$ *en*
- $defrelated(D, \alpha^{\mathcal{F}}(CS)) = \wp_{\emptyset}(D) \cup \{S_1 \cup S_2 \mid S_1 \in \wp_{\emptyset}(D), S_2 \in compl(D, \alpha^{\mathcal{F}}(CS))\}$.

Hieruit blijkt dat $\alpha^{\mathcal{F}}(CS)$ kan uitgedrukt worden in termen van D en $compl(D, \alpha^{\mathcal{F}}(CS))$. Dit leidt tot $\mathcal{DF}$ abstracte constraints, waarbij $\mathcal{DF}$ verwijst naar de optimalisatie van de $\mathcal{F}$ abstractie via het benutten van "definiteness" informatie.

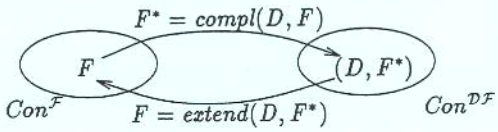
Definitie 5.5 ($\mathcal{DF}$ abstracte constraint)
Een $\mathcal{DF}$ abstracte constraint is ofwel $\perp$ ofwel (D, F^*) met $D \in \wp(Var)$ en $F^* \in \wp(\wp_{\emptyset}(Var \setminus D))$.

Definitie 5.6 ($\mathcal{DF}$ abstractie van een verzameling constraints)
Beschouw $CS \in Con^{\mathcal{C}}$. Dan $\alpha^{\mathcal{DF}}(CS) = \perp$ als $CS = \emptyset$, anders $\alpha^{\mathcal{DF}}(CS) = (D, F^*)$ waarbij D een deelverzameling (benadering) is van de verzameling uniek beperkte variabelen in CS, en $F^* = compl(D, \alpha^{\mathcal{F}}(CS))$.

De verzameling D kan bijvoorbeeld verkregen worden via de "definiteness" analyse ($\mathcal{D}$ analyse genoemd in de rest van de tekst), die ontwikkeld is door García de la Banda en Hermenegildo [5].

Voorbeeld 5.2
Beschouw $CS \equiv \{(X + Y + Z = 3 \wedge Y + Z = 2 \wedge T = f(Z)),\ (Y + Z = 5 \wedge T = 1 \wedge X = 3)\}$. *De verzameling uniek beperkte variabelen in CS is* $\{X\}$ *(de eerste constraint impliceert* $X = 1$*, de tweede bevat* $X = 3$*). De originele abstractie is* $\alpha^{\mathcal{F}}(CS) = \{\{X, Y, Z\}, \{X\}, \{Y, Z\}, \{T\}, \{Z, T\}, \{X, T\}, \{Y, T\}, \{X, Z, T\}, \{X, Y, T\}, \{Y, Z, T\}, \{X, Y, Z, T\}\}$. *Dit geeft* $\alpha^{\mathcal{DF}}(CS) = (\{X\}, \{\{Y, Z\}, \{T\}, \{Z, T\}, \{Y, T\}, \{Y, Z, T\}\})$. *De* $\mathcal{DF}$ *abstractie is veel compacter dan de* $\mathcal{F}$ *abstractie.*

Het verband tussen de $\mathcal{F}$ en $\mathcal{DF}$ abstractie van $CS \in Con^{\mathcal{C}}$ wordt geïllustreerd in Figuur 2, waarbij $F = \alpha^{\mathcal{F}}(CS)$ en $(D, F^*) = \alpha^{\mathcal{DF}}(CS)$ en $extend(D, F^*) = F^* \cup \wp_{\emptyset}(D) \cup \{S_1 \cup S_2 \mid S_1 \in \wp_{\emptyset}(D), S_2 \in F^*\}$. In tegenstelling tot de vorige optimalisatie (minimale abstractie) behoudt het benutten van "definiteness" informatie de precisie van de originele abstractie.



Figuur 2: Verband tussen de $Con^{\mathcal{F}}$ en $Con^{\mathcal{DF}}$ abstractie

Ook voor de $\mathcal{DF}$ abstracte constraints kunnen veel efficiëntere abstracte operaties gedefinieerd worden. Bij abstracte conjunctie b.v. wordt de "definiteness" informatie eerst berekend en dan gepropageerd op de F gedeeltes vóór die worden samengevoegd.

5.3 Combinatie

Beide optimalisaties, d.i. minimalisatie en het benutten van informatie over het uniek beperkt zijn van variabelen, kunnen gecombineerd worden, wat leidt tot een meer volledig en praktisch systeem om de beperktheid van variabelen te analyseren ($\mathcal{DM}$ analyse genoemd in de rest van de tekst). Een abstracte constraint is dan ofwel $\perp$ ofwel een paar (D, M^*). Hierin is D (een deelverzameling van) de verzameling van uniek beperkte variabelen; M^* is de verzameling van overblijvende mogelijk niet-vrije variabelen en mogelijke afhankelijkheden (die geen D-variabelen bevatten), in minimale vorm.

Voorbeeld 5.3
Beschouw $CS \equiv \{(X+Y+Z = 3 \wedge Y+Z = 2 \wedge T = f(Z)),\ (Y+Z = 5 \wedge T = 1 \wedge X = 3)\}$. *De verzameling uniek beperkte variabelen in CS is* $\{X\}$. *De originele abstractie is* $\alpha^{\mathcal{F}}(CS) = \{\{X, Y, Z\}, \{X\}, \{Y, Z\}, \{T\}, \{Z, T\}, \{X, T\}, \{Y, T\}, \{X, Z, T\}, \{X, Y, T\}, \{Y, Z, T\}, \{X, Y, Z, T\}\}$; $\alpha^{\mathcal{DF}}(CS) = (\{X\}, \{\{Y, Z\}, \{T\}, \{Z, T\}, \{Y, T\}, \{Y, Z, T\}\})$ *en de* $\mathcal{DM}$ *abstractie is* $\alpha^{\mathcal{DM}}(CS) = (D, M^*) = (\{X\}, \{\{Y, Z\}, \{T\}, \{Z, T\}, \{Y, T\}\})$. *De* $\mathcal{DM}$ *abstractie is nog compacter dan de* $\mathcal{DF}$ *abstractie. Het verschil met* $\alpha^{\mathcal{M}}(CS) = \{\{X\}, \{Y, Z\}, \{T\}, \{Z, T\}, \{Y, T\}\}$ *is dat in de* $\mathcal{DM}$ *abstractie* X *een uniek beperkte variabele is, die niet betrokken wordt in operaties op het* M^* *gedeelte.*

6 Uitbreidingen

Verschillende uitbreidingen van de basisanalyse uit sectie 4 worden aangehaald. Hierbij komen de volgende aspecten aan bod : het verhogen van de praktische toepasbaarheid of van de precisie van de analyse, het afleiden van bijkomende uitvoeringseigenschappen, en het behandelen van bijkomende constraint domeinen. We geven enkel een beknopte beschrijving.

6.1 Analyse van niet-genormaliseerde programma's

In niet-genormaliseerde programma's kunnen argumenten van procedure-oproepen en hoofdingen willekeurige termen zijn. Bovendien kunnen constraints zowel Herbrand als numerieke functoren bevatten, b.v. $X = f(Y + 1)$ hoeft niet langer genormaliseerd te worden naar $X = f(A) \wedge A - Y = 1$. Dit vereist het aanpassen van de abstractie van een constraint en van procedure-oproep en terugkeer. De gelijkheden tussen de corresponderende argumenten van een oproep en een procedure-hoofding zijn nu gelijkheden tussen willekeurige termen en niet langer eenvoudige hernoemingen van variabelen. De berekening bestaat dan uit het abstraheren van de gelijkheden en het toevoegen van deze abstracties, via abstracte conjunctie, aan de oproep- of uitgang-constraint van een procedure-definitie.

Het analyseren van niet-genormaliseerde programma's laat toe de precisie van de afgeleide informatie te verhogen. Een ander voordeel is de algemene bruikbaarheid van de uitgebreide analyse : zowel genormaliseerde als niet-genormaliseerde programma's kunnen behandeld worden. Het effect op de efficiëntie is niet eenduidig te voorspellen : enerzijds bevatten procedure-definities in niet-genormaliseerde programma's minder variabelen, wat leidt tot kleinere abstracte constraints en wat dus een gunstige invloed heeft op de efficiëntie; anderzijds omvat de analyse van niet-genormaliseerde programma's meer syntactisch verschillende procedure-oproepen, waardoor (zeker in het geval van recursieve procedures) de efficiëntie daalt (hieraan kan verholpen worden door een beter criterium te gebruiken voor het vergelijken van procedure-oproepen, zodat het genereren van nieuwe procedure-versies beperkt wordt).

6.2 Analyse van lineaire verschillen en ongelijkheden

Tot zover werden lineaire verschillen ($\neq$) en ongelijkheden ($>, \geq$) op dezelfde manier geabstraheerd als lineaire gelijkheden, en dus behandeld als symmetrische en transitieve relaties. Dit leidt tot onnauwkeurigheid, b.v. $X > Z \wedge Y > Z$ leidt *niet* tot een afhankelijkheid tussen X en Y en toch bevat de abstractie $\{X, Y\}$. De enige manier om meer preciese informatie af te leiden bestaat erin de vorm van de abstractie te veranderen. De abstractie moet minstens het soort constraint ($=, \neq, >, \geq$) en het teken van de coefficiënten bijhouden. Eventueel kan zelfs zo lang mogelijk de concrete waarde van de coefficiënten bewaard worden. Het is echter niet zo eenvoudig om een dergelijke uitgebreide abstractie van numerieke constraints te koppelen aan de abstractie voor unificatie constraints. Enerzijds moet informatie uitgewisseld worden tussen beide abstracties (b.v. $2X - 2Y = 0$ impliceert $X = Y$, wat het unificatie gedeelte kan beïnvloeden); anderzijds is ook een overkoepelende component nodig om afhankelijkheden voor te stellen die gecreëerd worden door een combinatie van numerieke en unificatie constraints (b.v. $X = f(A, B) \wedge A + B = T$ creëert

een afhankelijkheid tussen X en T). Dit staat open voor verder onderzoek.

6.3 Analyse van passieve constraints

Praktische CLP systemen maken vaak gebruik van partiële constraint oplossers. Complexe constraints worden uitgesteld tot genoeg informatie aanwezig is om ze op een eenvoudige manier te kunnen behandelen. Deze constraints worden *passieve* constraints genoemd. Een voorbeeld zijn de niet-lineaire numerieke constraints, die uitgesteld worden tot ze lineair geworden zijn (doordat bepaalde variabelen een unieke waarde hebben gekregen). De belangrijkste aspecten bij het analyseren van passieve constraints zijn :

1. het al of niet modelleren van het uitstellen/activeren van de constraints (dit impliceert het afwegen van precisie en complexiteit ten opzichte van efficiëntie);
2. het bepalen wat gedetecteerd moet worden : ofwel de *eerst mogelijke* activatie van de uitgestelde constraints, ofwel enkel de *zekere* activatie van de constraints (eventueel pas na de concrete activatie); dit hangt samen met het soort analyse (b.v. voor de analyse van vrijheidsgraden is het eerste van toepassing);
3. het opvangen van alle mogelijke gevallen bij de activatie van een constraint (b.v. afhankelijk van de waarde van X kan de niet-lineaire constraint $X * Y = Z$ herleid worden tot $0 = Z$ of $n * Y = Z$ met $n \neq 0$; in het eerste geval wordt Z niet-vrij, in het tweede geval blijft Z vrij maar onstaat een afhankelijkheid tussen Y en Z).

6.4 Afleiden van zekere en mogelijke faling

Voor bepaalde programma-optimalisaties, bijvoorbeeld het herordenen van constraints en procedure-oproepen, is het nodig te weten op precies welke programmapunten zeker of mogelijk faling optreedt. Een herordening is pas toegelaten als het faling-gedrag van het programma niet beïnvloed wordt. Een kleine uitbreiding van de analyse in sectie 4 geeft bijkomende informatie over zekere/mogelijke faling. Het uitbreiden bestaat uit het toevoegen van een speciaal element *Afail* aan het abstract domein en het toelaten van $\emptyset$ in de abstracte constraints. Hierbij geeft *Afail zekere* faling aan (overeenkomend met de constraint verzameling $\{false\}$ op een programmapunt tijdens de concrete uitvoering). De aanwezigheid van $\emptyset$ duidt op mogelijke faling (dit komt overeen met de aanwezigheid van *false* naast minstens één oplosbare constraint in de concrete verzameling van constraints); $\emptyset$ kan verkregen worden via de (aangepaste) abstracte conjunctie operatie. Belangrijk is ook dat faling niet verder gepropageerd wordt, maar enkel aangegeven wordt op het punt waar faling precies optreedt; dit wordt opgevangen via een bijkomende primitieve operatie *no_fail* (zie bij de volledige tekst).

6.5 Toevoegen van type informatie

In sectie 2 introduceerden we reeds een typering van variabelen als Herbrand, numeriek of niet-getypeerd. De combinatie van deze type informatie met informatie over het beperkt zijn van variabelen geeft een meer preciese beschrijving van de variabelen. Dit laat toe bijkomende programma-optimalisaties door te voeren, b.v. het omzetten van een constraint naar een toekenning waardoor de constraint oplosser niet meer opgeroepen moet worden. Een toekenning kan enkel gebeuren aan een variabele die vrij *en* niet-getypeerd is. Hoewel

het geven van een waarde aan een vrije numerieke variabele de oplosbaarheid van de verzameling (numerieke) constraints niet in gevaar kan brengen, zijn extra operaties nodig om de verzameling terug naar opgeloste vorm te transformeren. De constraint oplosser moet dus nog opgeroepen worden.

De aanwezigheid van type informatie leidt ook tot meer volledige informatie over faling ; naast faling ten gevolge van het geven van inconsistente waarden aan een variabele kan ook faling ten gevolge van een type conflict ontdekt worden.

Bij het afleiden of een variabele Herbrand, numeriek of niet-getypeerd is, is informatie over mogelijke afhankelijkheden tussen variabelen essentieel om voldoende nauwkeurigheid te bewaren. Zonder deze informatie zou elke variabele die niet-getypeerd is mogelijk getypeerd worden na abstracte conjunctie. Bijvoorbeeld, veronderstel dat X en Y vrije niet-getypeerde variabelen zijn; als X numeriek wordt door b.v. het toevoegen van $X + Z = 1$ en als Y kan afhangen van X, dan kan Y eventueel ook numeriek worden (merk op : Y blijft wel vrij). Anderzijds, als Y onafhankelijk is van X, dan blijft het niet-getypeerd zijn van Y behouden, wat belangrijk is met het oog op eventuele latere constraint specialisatie (een toekenning aan Y blijft mogelijk). Informatie over mogelijke afhankelijkheden is reeds aanwezig in de analyse van vrijheidsgraden, waardoor het afleiden van type informatie gemakkelijk kan geïntegreerd worden.

6.6 Analyse van andere constraint domeinen

Hoewel de analyse in sectie 4 in eerste instantie toegespitst werd op de Herbrand en numerieke constraint domeinen, kan ze ook uitgebreid worden naar andere constraint domeinen. Hiervoor moeten de noties van *vrij zijn* van een variabele (d.w.z. de variabele kan nog alle mogelijke waarden aannemen overeenkomstig zijn type), en van afhankelijkheden tussen variabelen, uitgebreid worden. In de volledige thesistekst wordt het toevoegen van het PrologIII lijst domein beschreven (PrologII is een specifieke CLP taal).

7 Resultaten

Een prototype van de analyses in secties 4 en 5, samen met een aantal van de uitbreidingen in sectie 6, werd geïmplementeerd binnen het abstracte interpretatie systeem PLAI [9] (geschreven in SICStus Prolog). Voor het benutten van "definiteness" informatie werd een koppeling gemaakt met de $\mathcal{D}$ analyse van García de la Banda en Hermenegildo [5]; dit gebeurde in samenwerking tussen K.U.Leuven en U.P.Madrid in het kader van het ESPRIT project PRINCE. Met het prototype werd een verscheiden collectie van CLP programma's geanalyseerd. De besluiten van de evaluatie worden kort samengevat.

7.1 Efficiëntie

De combinatie van minimalisatie en het benutten van "definiteness" informatie ($\mathcal{DM}$ analyse) geeft vrij goede resultaten, zowel wat het tijds- als geheugengebruik betreft. De analysetijden voor de beschouwde niet-genormaliseerde programma's schommelen tussen 0.019 en 26.019 secondes (SUN Sparc 2, SICStus fastcode); het gemiddeld geheugenverbruik bedraagt 4.033 MByte, met een maximum van 10.373 Mbyte. Voor genormaliseerde

programma's liggen de analysetijden tussen 0.030 en 145.093 seconden en het gemiddeld geheugenverbruik is 5.048 Mbyte, met een maximum van 34.054 Mbyte; voor één programma werd geen resultaat bekomen binnen een redelijk tijds- en geheugenverbruik.
Bij de $\mathcal{DM}$ analyse worden de "definiteness" ($\mathcal{D}$) analyse en de minimale analyse van vrijheidsgraden ($\mathcal{M}$) uitgevoerd volgens het coroutine mechanisme (voor elke abstracte operatie gebeurt de $\mathcal{D}$ analyse vóór de $\mathcal{M}$ analyse). Voor sommige programma's leiden elk van beide analyses tot een verschillend aantal iteraties over de procedure-definities. Bij de gecombineerde $\mathcal{DM}$ analyse wordt één analyse gedwongen mee te itereren met de andere, zodat de uitvoeringstijd van de $\mathcal{DM}$ analyse in een aantal gevallen toch groter wordt dan de som van de tijden van de afzonderlijke $\mathcal{D}$ en $\mathcal{M}$ analyses.

Wat de impact van normalisatie betreft, is de analyse van de niet-genormaliseerde programma's in 57.9% van de gevallen sneller dan de analyse van de genormaliseerde programma's. Vooral voor de grotere programma's leidt het *niet* normaliseren tot betere resultaten. Het verschil in geheugengebruik is minder uitgesproken. De reden is dat het SICStus systeem bij uitbreiding van de uitvoeringsstructuren steeds een vaste hoeveelheid geheugen toevoegt.

7.2 Nauwkeurigheid

De nauwkeurigheid werd enkel nagegaan voor de $\mathcal{D}$, $\mathcal{M}$ en $\mathcal{DM}$ analyses van niet-genormaliseerde programma's. Voor de $\mathcal{F}$ en $\mathcal{DF}$ analyses kan de precisie enkel groter zijn (minimalisatie leidt namelijk tot een mogelijk verlies van precisie); voor *genormaliseerde* programma's is een slechtere nauwkeurigheid mogelijk. Onnauwkeurigheid voor de $\mathcal{D}$ analyse of het $\mathcal{D}$ gedeelte van de gecombineerde $\mathcal{DM}$ analyse betekent dat het uniek beperkt zijn van sommige variabelen niet ontdekt wordt. In het geval van de $\mathcal{M}$ analyse en het $\mathcal{M}$ gedeelte van de $\mathcal{DM}$ analyse betekent onnauwkeurigheid dat bepaalde variabelen als mogelijk niet-vrij beschouwd worden waar ze dat in de concrete uitvoering wel nog zijn. De $\mathcal{D}$ analyse levert een gemiddelde nauwkeurigheid van 95.7% op, de $\mathcal{M}$ analyse 94% en de $\mathcal{DM}$ analyse 89.7%.

Er zijn drie mogelijke oorzaken van onnauwkeurigheid : (1) het gebrek aan informatie over termstructuren, (2) het behandelen van niet-lineaire constraints en (3) het feit dat in de implementatie *primitieve* constraints geabstraheerd worden en hun abstracties samengevoegd worden via abstracte conjunctie, in plaats dat direct een volledige constraint geabstraheerd wordt. Theoretisch gezien kan onnauwkeurigheid ook ontstaan door minimalisatie van de abstracties en door een ruwe benadering van verschillen en ongelijkheden. Dit kwam echter niet voor bij de beschouwde programma's.

7.3 Gebruik van de informatie

De afgeleide informatie kan gebruikt worden voor een brede waaier van programma-optimalisaties. In [7, 8] wordt aangetoond dat deze optimalisaties de efficiëntie van de uitvoering sterk kunnen verbeteren.

Eén van de optimalisaties bestaat erin constraints te transformeren naar een eenvoudige test of toekenning. Bijvoorbeeld, een numerieke gelijkheid wordt een toekenning aan een variabele X, indien X vrij en niet-getypeerd is in de oproep-constraint van de gelijkheid

en indien de overige variabelen in de gelijkheid alle beperkt zijn tot een unieke waarde. Zo wordt het oproepen van de constraint oplosser vermeden. Zelfs als de constraint oplosser toch nog moet opgeroepen worden, kan dergelijke informatie gebruikt worden om speciale instructies te voorzien die leiden tot een efficiëntere uitvoering. Informatie over het uniek beperkt zijn van variabelen laat toe te detecteren of niet-lineaire constraints reeds lineair zijn op het moment van hun evaluatie. Dit vermijdt het genereren van code voor het uitstellen/activeren van de constraints. Als een bepaald argument van een procedure-definitie beperkt is tot een unieke waarde en de structuur van dat argument verschilt in de verschillende procedure-definities, kan slechts één definitie slagen. De creatie van een keuzepunt wordt dan overbodig. Vrije niet-getypeerde argumenten daarentegen kunnen niet gebruikt worden voor het kiezen van één bepaalde definitie. De uitvoering van een programma kan ook verbeterd worden door primitieve constraints te herordenen; het idee daarbij is dat de meest beperkende constraints best eerst uitgevoerd worden zodat faling zo vroeg mogelijk optreedt. Deze optimalisatie steunt op informatie over het vrij zijn en het uniek beperkt zijn van variabelen. Een andere belangrijke klasse van optimalisaties steunt op het onafhankelijk zijn van constraints of procedure-oproepen. Dit betekent dat de constraints/oproepen elkaar niet kunnen beïnvloeden in de zin dat het toevoegen van de constraints (de constraints zelf of de constraints uit de oproepen) aan de tot dan toe verzamelde set van constraints geen aanleiding kan geven tot faling. De informatie over mogelijke faling (sectie 6.4) laat toe te detecteren wanneer constraints/oproepen zeker onafhankelijk zijn. Dit kan dan gebruikt worden voor het herordenen of voor het parallel uitvoeren van de constraints/oproepen, en voor intelligente backtracking. Tenslotte bevatten de constraints die tijdens de uitvoering bijgehouden worden door de constraint oplossers, vaak overbodige variabelen. Informatie over het uniek beperkt zijn van variabelen en over mogelijke afhankelijkheden tussen variabelen maakt deel uit van wat nodig is om dergelijke variabelen te verwijderen.

8 Besluit

Deze thesis levert een bijdrage tot de globale analyse van constraint logische programma's. Het doel hierbij is het afleiden van uitvoeringseigenschappen die nuttig zijn bij codegeneratie. Een dergelijke optimaliserende vertaling van CLP programma's zou de efficiëntieproblemen van huidige CLP systemen moeten oplossen of althans sterk verminderen.
De analyse werd geformaliseerd in termen van abstracte interpretatie. Hierbij werd gebruik gemaakt van een bestaand raamwerk voor de analyse van logische programma's, dat reeds aangepast werd voor CLP. Binnen dat raamwerk werd een specifieke analyse ontworpen voor het afleiden van vrijheidsgraden in constraints en het nagaan van mogelijke constraint interactie. De correctheid van deze analyse werd bewezen. Daarnaast werden verschillende optimalisaties en uitbreidingen van de basisanalyse ontwikkeld.
Uit dit werk kwamen een aantal algemene aspecten omtrent het analyseren van CLP programma's naar voor. De basis van elke CLP analyse bestaat uit het nauwkeurig abstraheren van constraint-implicatie (afgeleide constraints). Hierbij moet nauwkeurigheid afgewogen worden ten opzichte van efficiëntie. De complexiteit van de analyse is groter dan in het geval van LP : enerzijds is er de intrinsiek hogere complexiteit van de algoritmes voor het oplossen van constraints; anderzijds moet ook rekening gehouden worden met de interactie

tussen verschillende constraint domeinen.
Verder onderzoek is nodig om de ontwikkelde analyses te integreren in een optimaliserende vertaler [8]. Belangrijke aspecten hierbij zijn (1) het detecteren waar welke optimalisaties mogelijk zijn, steunend op de afgeleide informatie, en (2) het nagaan en opvangen van niet-triviale interacties tussen verschillende optimalisaties (het doorvoeren van één optimalisatie kan een andere optimalisatie op een andere plaats in het programma uitsluiten). De verhoogde efficiëntie, die door een verbeterde vertaling kan verkregen worden, zou, samen met de expressieve kracht van CLP talen, moeten leiden tot een grotere erkenning van CLP.

Referenties

[1] M. Bruynooghe. A practical framework for the abstract interpretation of logic programs. *Journal of Logic Programming*, 10(2):91–124, Feb. 1991.

[2] P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *Journal of Logic Programming*, 13(2&3):103–179, 1992.

[3] V. Dumortier and G. Janssens. Towards a practical full mode inference system for CLP(H,N). In P. Van Hentenryck, editor, *Proceedings of the 11th International Conference on Logic Programming*, pages 569–583, Italy, June 1994. MIT Press.

[4] V. Dumortier, G. Janssens, M. Bruynooghe, and M. Codish. Freeness analysis in the presence of numerical constraints. In D. S. Warren, editor, *Proceedings of the 10th International Conference on Logic Programming*, pages 100–115, Budapest, Hungary, June 1993. MIT Press.

[5] M. García de la Banda and M. Hermenegildo. A Practical Approach to the Global Analysis of CLP Programs. In D. Miller, editor, *Proceedings of the 10th International Logic Programming Symposium*, pages 437–455, Vancouver, Canada, Oct. 1993. MIT Press.

[6] J. Jaffar and M. J. Maher. Constraint Logic Programming : a survey. *Journal of Logic Programming*, 19/20:503–581, May/July 1994.

[7] N. Jørgensen, K. Marriott, and S. Michaylov. Some global compile-time optimizations for CLP(R). In V. Saraswat and K. Ueda, editors, *Proceedings of the 1991 International Symposium on Logic Programming*, pages 420–434, San Diego, USA, Oct. 1991. MIT Press.

[8] K. Marriott, H. Søndergaard, P. J. Stuckey, and R. H. Yap. Optimizing compilation for CLP($\mathcal{R}$). In *Proceedings of the 17th Annual Computer Science Conference*, Christchurch, New Zealand, Jan. 1994.

[9] K. Muthukumar and M. Hermenegildo. Compile-time Derivation of Variable Dependency Using Abstract Interpretation. *Journal of Logic Programming*, 13(2&3):315–347, July 1992.