KU LEUVEN

ARENBERG DOCTORAL SCHOOL
Faculty of Engineering Science

Higher order modelling for first order solvers

Ingmar Dasseville

Supervisors:
Prof. dr. M. Denecker
Prof. dr. ir. G. Janssens
(co-supervisor)

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor of Engineering Science (PhD): Computer Science

June 2019

Higher order modelling for first order solvers

Ingmar DASSEVILLE

Examination committee:
Prof. dr. ir. O. Van der Biest, chair
Prof. dr. M. Denecker, supervisor
Prof. dr. ir. G. Janssens, supervisor
(co-supervisor)
Prof. dr. ir. T. Schrijvers
Prof. dr. G. B. Ianni
(University of Calabria)
Prof. dr. M. Hanus
(University of Kiel)

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor of Engineering Science (PhD): Computer Science

June 2019

© 2019 KU Leuven – Faculty of Engineering Science
Uitgegeven in eigen beheer, Ingmar Dasseville, Celestijnenlaan 200A box 2402, B-3001 Leuven (Belgium)

Alle rechten voorbehouden. Niets uit deze uitgave mag worden vermenigvuldigd en/of openbaar gemaakt worden door middel van druk, fotokopie, microfilm, elektronisch of op welke andere wijze ook zonder voorafgaande schriftelijke toestemming van de uitgever.

All rights reserved. No part of the publication may be reproduced in any form by print, photoprint, microfilm, electronic or any other means without written permission from the publisher.

Acknowledgements

At the end of this journey, I should thank a few people. Thank you Marc for giving me the opportunity to do my doctorate. Thank you Gerda for joining me as a co-promotor. Thank you Maurice, for being the voice of reason in difficult discussions. Thank you Bart, Tom, Giovambattista and Michael for your work as the jury and your suggestions for improvement.

Thank you to all other people who I worked with in the group: Broes, Bart, Pieter, Jo, Matthias, Ruben, Laurent, Tim, Simon and Pierre.

Thank you to the improvisation theatre community to keep me sane in times of high stress. Thank you to my friends and family for supporting me over the years. Thank you Jeti for loving me, despite my endless stream of complaints, and for keeping me motivated throughout the years.

Abstract

Many common computational problems can be posed as model expansion problems, where the task is to find a solution satisfying a set of constraints. Many different paradigms and approaches exist to solve these problems. Likewise, there are many languages to express these problems in. Some of these are standardized, such as Answer Set Programming (ASP) or Minizinc. For some languages there is only one system, like the FO($\cdot$) language. The language one chooses to express the problem in, has a high impact on both readability of the modelling and the efficiency of an automated solver.

In this thesis we explore higher-order language constructs for modelling languages. These are generally considered very readable yet very inefficient, but in some cases this tradeoff is not so strict. This thesis is divided into three parts.

The first part of the thesis focusses on the semantics of higher-order languages. Chapter 2 extends FO($\cdot$) with templates as second-order definitions. Templates were already known as language constructs in other languages such as ASP, but their semantics were given through a rewriting process rather than giving a meaning to the templates themselves.

Chapter 3 explores how logics are built and introduces infons as the basis for defining the semantics of an arbitrary language construct. Using this approach, we introduce a uniform way to extend languages with (higher-order) language constructs and define a higher-order logic with definitions.

The second part of the thesis shows how existing ASP solvers can be used to solve model expansion problems posed in a higher-order language. Chapter 4 explains a translation from Programming Computable Functions (PCF) to Answer Set Programming. It then explains how this translation can handle extensions of PCF to obtain a higher-order modelling language which can be translated to ASP.

Chapter 5 explains the workflow of the Functional Modelling System (FMS). This system is a prototype implementation of the language described in Chapter 4. It describes techniques for optimising modellings and how the translation phase fits in a larger system.

Even the best languages are easier to use with good supportive software. In the last part of the thesis, we shift our focus away from higher-order features and introduce some tooling that was developed for the IDP3 system. Chapter 6 introduces a web-based IDE for IDP3. This IDE enables new users to easier write theories and get started with IDP.

Chapter 7 introduces an interactive configuration tool which was developed to show how specifications written in Decision Model and Notation (DMN) can be used to do more than just draw conclusions from premises. It enables the user to interactively explore the solution space of a configuration problem.

Beknopte samenvatting

Veel vaakvoorkomende computationele problemen kunnen voorgesteld worden als modelexpansie problemen, dit probleem bestaat uit het zoeken naar een oplossing die voldoet aan een verzameling voorwaarden. Veel verschillende aanpakken, paradigma's en talen worden gebruikt om dit soort problemen aan te pakken. Sommige zijn gestandariseerd zoals Answer Set Programming (ASP) of Minizinc. Andere talen hebben maar één implementatie zoals de FO($\cdot$) taal. De gekozen taal heeft een grote impact op zowel de leesbaarheid van de modellering en de efficiëntie van een geautomatiseerde oplosser.

In deze thesis onderzoeken we hogere-orde taalconstructies voor modelleertalen. Deze constructies worden in het algemeen als zeer leesbaar maar zeer inefficiënt beschouwd, maar in sommige gevallen is dit niet zo strikt. Deze thesis bevat 3 onderdelen.

Het eerste gedeelte van de thesis focust op de semantiek van hoger-orde talen. Hoofdstuk 2 breidt FO($\cdot$) uit met templates en tweede-orde definities. Templates zijn een gekende taalconstructie in andere talen zoals ASP, waar hun semantiek is gedefinieerd als een herschrijfproces. Hoofdstuk 2 geeft een betekenis aan de templates zelf in de context van FO($\cdot$).

Hoofdstuk 3 gaat dieper in op de bouw van logica's en introduceert infons als de basis voor het definiëren van de semantiek van een taalconstructie. Via deze aanpak introduceren we een uniforme manier om talen uit te breiden met nieuwe (hoger-orde) taalconstructies en definiëren we zo een hogere-orde logica met inductieve definities.

Het tweede gedeelte van de thesis toont hoe bestaande ASP systemen gebruikt kunnen worden om modelexpansie problemen op te lossen die gesteld worden hogere-orde talen. Hoofdstuk 4 legt uit hoe Programming Computable Functions (PCF) naar Answer Set Programming kan vertaald worden. Verder wordt uitgelegd hoe dit principe kan gebruikt worden om uitbreidingen van PCF te ondersteunen zodat we een hogere-orde modelleertaal verkrijgen die vertaald

kan worden naar ASP.

Hoofdstuk 5 legt de algemene werking uit van het Functional Modelling System (FMS). Dit systeem is een prototype implementatie van de taal die beschreven is in Hoofdstuk 4. Het beschrijft enkele technieken voor het optimaliseren van modelleringen en hoe het vertaalproces past in een groter systeem.

Zelfs de beste talen zijn makkelijker om te gebruiken met ondersteunende software. In het derde en laatste gedeelte van de thesis veranderen we de focus van hoger-orde talen naar ondersteunende software voor het IDP3 systeem. Hoofdstuk 6 introduceert een web-gebaseerde IDE voor IDP4. Deze IDE laat nieuwe gebruikers toe om makkelijker modelleringen te schrijven en de beginselen van IDP3 te leren.

Hoofdstuk 7 introduceert een tool voor interactieve configuratieproblemen. Deze tool werd ontwikkeld om aan te tonen hoe specificaties die geschreven zijn in Decision Model and Notation (DMN) gebruikt kunnen worden om meer te doen dan conclusies te trekken uit gegevens. Het laat namelijk toe om interactief de oplossingsruimte van een configuratieprobleem te exploreren.

Contents

Abstract iii

Beknopte samenvatting v

Contents vii

1 Introduction 1

I Language Building 5

2 Semantics of templates in a compositional framework for building logics 7

2.1 Introduction 7

2.2 Related Work 8

2.3 Preliminaries 10

2.4 Well-founded and stable semantics for $\mathcal{L}$-rule sets 13

2.5 Compositional framework for building logics with definitions . . 15

2.6 Templates 20

2.7 Conclusion 26

3 A Compositional Higher-Order Logic with Definitions 27

3.1 Introduction 27
3.2 Related Work 28
3.3 Preliminaries 30
3.4 A typed higher order logic $\mathbb{L}$ with (nested) definitions 34
3.5 Conclusion 44

II Implementing a Lambda Calculus in Answer Set Programming 45

4 Translating Programming Computable Functions to Answer Set Programming 47

4.1 Introduction 47
4.2 Programming Computable Functions 48
4.3 Answer Set Programming 52
4.4 Translation 54
4.5 Applications 63
4.6 Towards a more expressive language 64
4.7 Typing 73
4.8 Declaring non-defined symbols 75
4.9 Conclusion 76

5 FMS: Functional Programming as a Modelling Language 77

5.1 Introduction 77
5.2 Language 81
5.3 General Workflow 85
5.4 Optimisations 91
5.5 Dependencies and availability 95
5.6 Conclusion 95

III Tooling for IDP3 **97**

6 A web-based IDE for IDP **99**

6.1 Introduction . . . 99
6.2 The IDP system . . . 100
6.3 Overview of the IDE . . . 102
6.4 Other IDEs . . . 108
6.5 Integration in other web applications . . . 110
6.6 Conclusion . . . 111

7 Combining DMN and the KB Paradigm for Flexible Decision Enactment **113**

7.1 Introduction . . . 113
7.2 Background . . . 114
7.3 The Running Example . . . 116
7.4 From Decision Modelling to Enactment . . . 118
7.5 Enactment Scenarios . . . 120
7.6 Major Contributions and Future Work . . . 124
7.7 Conclusion . . . 125

8 Conclusion **127**

A Defining Letrec in terms of Fixpoints **129**

Bibliography **133**

List of Publications **145**

1 Introduction

A lot of people, myself included, turn to a computer whenever they have a problem which seems to require more work than a few minutes. They hope that an automated way of solving the problem will save them time. However, the time needed to explain the problem to the computer needs to be taken into account to judge whether it is worth to automate solving the problem. If we are able to make it more efficient to explain a problem to a computer, we can increase the gains that the computer is able to give us.

An important class of problems which routinely need to be solved by a computer, are model expansion problems. A model expansion problem is a general class of problems where a solution for a problem needs to be found given a set of constraints. Rostering problems are examples of model expansion problems where a roster needs to be found under the constraints that people should have enough holidays, enough people with the right degrees should be present at any moment, ...

Optimisation problems are closely related to model expansion problems. These add an optimisation criterion to the problem. The new goal is to find a solution, which still needs to obey the set of constraints, but on top of this it should also optimise this criterion. Rostering problems are often presented as an optimisation problem where one needs to minimise the number of needed people or resources.

Systems There are many flavours of languages and automated systems suitable for expressing model expansion problems. This thesis will refer to such languages as modelling languages. Some of the most prominent languages are Minizinc [97], ASP-core [17] (the common base language for Answer Set Programming) and A Mathematical Programming Language (AMPL) [56]. There are multiple systems that accept a specification in such a language and output a solution.

Next to the standardized languages, many systems provide their own language, dialect or extensions. The work in this thesis makes use of such a system, i.e., the IDP3 system [37] which provides the FO(·) language. This language encompasses first-order logic and some extensions such as arithmetic, sorts and inductive definitions.

This wide range of systems is available, but there is still something lacking. Although they aim to provide a simple way of expressing model expansion problems, they are in practice mainly used by highly specialized people. This means there is still room for improvement so any programmer or even non-programmers are able to use these systems. One such aspect which could be improved is the expressivity of the modelling languages.

Expressivity An increased expressivity of a language generally leads to shorter and more readable expressions. This leads to a more intuitive modelling experience which would attract more people to use the system. However, some trade-offs in expressivity need to be made to ensure that a computer can still find solutions in a feasible time span. Therefore, higher-order language features are generally unavailable in modelling languages. These features would nevertheless often make it easier for an end-user to express his problem.

A smart handling of some higher order language features could lead to more expressivity without impairing the efficiency of the language. One of the most useful higher order language features is *templates*. These are higher order expressions which have the guarantee that they can be eliminated from the modelling in an easy way so that an equivalent fully first-order modelling can be obtained.

Of course not all higher order features are that simple and not every conceivable higher order expression can be added to a language without fundamentally altering the available solvers. So it is interesting to investigate what is possible, both from a theoretical and practical point of view.

Using this approach we can use first order solvers to solve problems which are posed in a higher order modelling language. This is exactly the main topic (and title) of this thesis.

Contributions and Structure of the Text

This thesis hopes to contribute to the insights about modelling languages. It includes three parts. The first two parts focus on the higher order languages described above. The first part from a semantic point of view, the second part

from an implementation and translation point of view. The third part focusses on new tooling for IDP3.

The first part of the thesis is about modelling languages and their semantics. Languages should have clear non-ambiguous semantics which correspond to our intuitions. In the past, defining language extensions has proven to be a difficult task. In this part we give some more elegant semantics to existing language extensions and provide a framework which should lessen the work of introducing new language extensions.

- **Chapter 2** introduces templates as an extension to the FO(·) family of languages. Templates are higher-order definitions, which can be eliminated by a simple rewriting process. These templates were previously known as syntactic constructs for ASP, but the semantics were only defined through a rewriting process. In this chapter we identify them as a proper language construct with semantics. This work was published in the journal "Theory and Practice of Logic Programming", volume 15 [35].
- **Chapter 3** provides a framework to build logics based on the lambda calculus, and integrated with traditional well-founded definitions. We introduce infons as the essential object conveying the meaning of an expression. It is a more general logic than the logic which was introduced in Chapter 2. This work was published in the Technical Communications of the 32nd International Conference on Logic Programming. [34].

The second part of the thesis focuses on the relation between lambda calculi and logic programs and how a lambda calculus can be used as a modelling language. This chapter is more practically oriented as a real system has been implemented to demonstrate the feasibility of the approaches presented in this part.

- **Chapter 4** contains a translation algorithm from Programming Computable Functions to Answer Set Programming. This translation allows us to reuse existing technology as a basis for developing new modelling languages. This work was presented at the international Workshop on Functional and (constraint) Logic Programming 2018.
- **Chapter 5** explains a prototype system FMS which uses the techniques of Chapter 4 to explain the implementation of a modelling language with characteristics similar to Chapter 3. It explains a high-level overview of the system and gives a step-by-step explanation of the normal workflow. This work was presented at the international Workshop on Functional and (constraint) Logic Programming 2018.

The third part of the thesis deviates from the subject of higher order languages and explains some of the tooling that was developed for IDP3. Even the best languages benefit from some support when writing in them. For this reason, a new IDE was developed for IDP3. This IDE has been actively used the past years in courses where IDP3 is used. The second tool that is introduced in this part is a tool for interactive configuration problems. This tool was developed to show how the knowledge base paradigm can be applied to Decision Model and Notation (DMN) specifications.

- **Chapter 6** introduces a new development environment for writing IDP3 specifications. This environment can be run like a regular IDE for working on a personal machine, but can also be used as a web server to provide a simple way for new users to try out IDP3 online without the need to install any software. This work was presented at the International Workshop on User-Oriented Logic Programming 2015 [31].
- **Chapter 7** introduces a new tool for exploring IDP3 specifications as an interactive configuration problem. It was build to show how specifications written in Decision Model and Notation can be used to do more than just draw conclusions from premises. This work was presented on the 10th International Web Rule Symposium and the developed tool won the RuleML Challenge Award 2016 on that conference [33].

Part I

Language Building

About This Part

Building a language is a hard task. From a bird's eye view it consists of two steps. The first step consists of defining the syntax of the language. This roughly corresponds to: "What does the language look like?". Defining a syntax is mostly done using a grammar, often in Backus-Naur format. This defines which expressions are part of the language, but doesn't attach any meaning to these expressions.

The second step consists of defining the semantics of the language. This roughly corresponds to : "What does an expression in the language mean?". Defining semantics is a task which can be far more controversial. There are a myriad of different notions of semantics. The meaning of traditional programming languages is usually defined using *operational semantics*. Operational semantics associates some way of executing the language to the expressions in the language.

For purely declarative languages, operational semantics are less appropriate. These languages have no notion of execution, instead they describe a set of possible worlds. In these cases, *model semantics* are more appropriate. Model semantics associate a set of models with every expression. This can also be characterised by a function mapping structures (solutions) to true or false.

For some applications a more refined semantics is interesting, in which partiality of information is taken into account. *Three-valued model semantics* can indicate that $p \vee q$ is true if p is true, even when the value of of q is not determined yet.

This first part of the thesis consists of two chapters on building languages with model semantics. In Chapter 2 we generalize the FO$(\cdot)$ language and introduce templates. Semantics of templates have been defined in the past through a rewriting algorithm. In this chapter we use pure model semantics and we argue this gives more insight into the nature of the language construct.

In Chapter 3 we introduce an even more general language based on the lambda calculus. A major contribution of this chapter is the introduction of infons, which provides a more structured way for defining model semantics of languages.

2 Semantics of templates in a compositional framework for building logics

A first version of the work in this chapter was published as a journal paper in Theory and Practice of Logic Programming [35] and presented at the International Conference on Logic Programming 2015.

2.1 Introduction

Declarative specification languages have proven to be useful in a variety of applications, however sometimes parts of specifications contain duplicate information. This commonly occurs when different instantiations are needed of an abstract concept. For example, in an application, we may have to assert of multiple relations that they are an equivalence relation. In most current logics, the constraints (e.g., reflexivity, symmetry and transitivity) need to be reasserted for each relation.

In the early days of programming, imperative programming languages suffered from a similar situation where code duplication was identified as a problem. The first solution used for this was the use of macros, where a syntactical replacement was made for every instantiation of the macro. Later on, using subroutines was identified as a way to reduce code duplication with a more intuitive semantics. For specification languages, the analog for macros was introduced (e.g., in ASP), most often called *templates*. These allow to define a concept and to instantiate it multiple times, without making the language more computationally complex. Asserting that the two relations `P` and `Q` are equivalence relations could be done using a template `isEqRelation` as can be seen in Listing 2.1.

Listing 2.1: P and Q are asserted as equivalence relations

```
{
∀F: isEqRelation(F) ←
  (∀a : F(a,a)) ∧
  (∀a ∀b : F(a,b) ⇔ F(b,a)) ∧
  (∀a ∀b ∀c : (F(a,b) ∧ F(b,c)) ⇒ F(a,c)).
}
isEqRelation(P).
isEqRelation(Q).
```

In existing treatments of templates, their semantics is given in a transformational way, essentially by translating them away as if they were macros. This approach has its limitations. A more declarative view is that templates are higher order definitions. This allows for a much more general treatment. In some interesting cases, these higher order template definitions are recursive (see Listing 2.4). In others, like the template symbol $tc(P, Q)$ specifying P as the transitive closure of Q, the definiens is itself an inductive definition (see Listing 2.2) nested in the template definition of tc.

The goal of this work is to introduce a declarative template mechanism for the language FO($\cdot$). This logic posesses an expressive first order definition construct in the form of rules under well-founded semantics which was shown suitable to express informal definitions of the most common types [44]. We want to extend FO($\cdot$)'s definition construct to nested higher order definitions.

In the first part of this chapter, we present a compositional framework for building an infinite class of logics. This framework specifies a principled way for building rule formalisms under well-founded and stable semantics from arbitrary logics, and ways to compose and nest arbitrary language constructs including higher order symbols, rule sets and aggregates. In the second part we use this framework to build a template formalism. As a last contribution we show that under suitable conditions, the standard approach of templates as rewriting macros also works in this formalism, thus recovering the results of existing approaches.

2.2 Related Work

Abstraction techniques have been an important area of research since the dawn of programming [111]. A lot of this work is syntax-oriented, and based on the notion that abstractions can be expanded to a language without them [92].

Popular programming languages such as C++ consider templates as a keystone for abstractions [94]. Within the ASP community, work by Ianni et al. [18] and Baral et al. [8] introduced concepts to support composability, called templates and macros respectively. The key idea is to abstract away common constructs through the definition of generic 'template' predicates. These templates can then be resolved using a rewriting algorithm.

More formal attempts at introducing more abstractions in ASP were made. Dao-Tran et al. [28] introduced modules which can be used in similar ways as templates but has the disadvantage that his template system introduces additional computational complexity, so the user has to be very careful when trying to write an efficient specification.

Previously, meta-programming [2] has also been used to introduce abstractions, for example in systems such as HiLog [20]. One of HiLog's most notable features is that it combines a higher-order syntax with a first-order semantics. HiLog's main motivation for this is to introduce a useful degree of second order yet remain decidable. While decidability is undeniably an interesting property, the problem of decidability already arises in logic programs under well-founded or stable semantics, certainly with the inclusion of inductive definitions: the issue of undecidability is not inherent to the addition of template behavior. As a result, in recent times deduction inference has been replaced by various other, more practical inference methods such as model checking, model expansion, or querying. Furthermore, for practical applications, we impose the restriction of stratified templates for which an equivalent first-order semantics exists.

An alternative approach is to see a template instance as a call to its definition in another theory, using another solver as an oracle. An implementation of this approach exists in HEX [51, 50]. This implementation however suffers from the fact that the different calls occur in different processes. As a consequence, not enough information is shared which hurts the search. This is analog to the approach presented by Tasharrofi [119], where a general approach to modules is presented. A template would be an instance of a module in this framework, however the associated algebra lacks the possibility to quantify over modules.

Previous efforts where made to generalize common language concepts, such as the work by Lifschitz [87] who extended logic programs to allow arbitrary nesting of conjunction $\wedge$, disjunction $\vee$ and negation as failure in rule bodies. The nesting in this chapter is of very different kind, by allowing the full logic, including definitions itself, in the body.

In more recent work, Charalambidis et al. have developed semantics for higher-order logic programs [19]. These higher-order logic programs allow us to express some of the templates introduced in this chapter. However, it is not obvious

how this approach can be extended to encompass a full logic in which more language constructs such as universal quantification can be used. However, Charalambidis's higher-order semantics is better suited for implementations than the one in this chapter. Combining these ideas with the one presented there could lead to interesting new work.

2.3 Preliminaries

Symbols. We assume an infinite supply of (typed) symbols. A vocabulary Σ is a set of (typed) symbols. For each symbol σ, $\tau(\sigma)$ is its type. For a tuple $\bar{\sigma}$, $\tau(\bar{\sigma})$ denotes the tuple of types.

An untyped logic is one with a single type. But for the purposes of this chapter, it is natural to use at least a simple form of typing, namely to distinguish between first order symbols and the second order (template) symbols. We distinguish between base types (some of which may be interpreted, e.g., $\mathbb{B}$, $\mathbb{Z}$) and composite types. A simple type system that suffices for this chapter consists of the following types:

- base types δ and $\mathbb{B}$; δ represents the domain of the terms in the theory;
- first order types: n-ary predicate types $\delta^n \to \mathbb{B}$ and function types $\delta^n \to \delta$. As usual, propositional symbols and constants are predicate and function symbols of arity $n = 0$.
- second order types: n-ary predicate types $(\tau_1, \ldots, \tau_n) \to \mathbb{B}$ with each τ_i a first order type or δ.

This is the type system that we have in mind in this chapter. It suffices to handle untyped first order logic and second order predicates (no second order functions are needed). However, the framework below is well-defined for much richer type systems (including higher order types, type theory).

While we need function types, we avoid function symbols to simplify the presentation. However, the framework can be easily extended with them. Moreover, for logics such as FO, n-ary function symbols can be emulated using their n+1-ary graph predicate symbols. E.g., one can represent a unary function $f(x) = y$ as a binary relation $f(x, y)$ with an implicit functional constraint between the arguments.

(Partial) values. For interpreted base types, there is a fixed domain of values. E.g., the domain of the boolean type $\mathbb{B}$ is $\mathcal{T}wo = \{\mathbf{t}, \mathbf{f}\}$. For other base types τ,

the domain of values is chosen freely. For composite types, the set of values is constructed from the values of the base types.

For the simple type system above, the values of all types are determined by the choice of the domain associated with δ. For any domain D, we can define the domain τ^D of any type τ as follows:

- $\delta^D = D$, $\mathbb{B}^D = \mathcal{T}wo$
- first order predicates: $(\delta^n \to \mathbb{B})^D$ is the set of all functions from D^n to $\mathcal{T}wo$ (or equivalently, the set of all subsets of D^n). For first order functions, $(\delta^n \to \delta)^D$ is the set of all functions from D^n to D.
- second order predicates: $((\tau_1, \ldots, \tau_n) \to \mathbb{B})^D$ is the set of all functions from ${\tau_1}^D \times \cdots \times {\tau_n}^D$ to $\mathcal{T}wo$.

To define the semantics of inductive definitions, partial values for predicates are essential (since only predicates are defined in the logics of this chapter, we do not introduce partial values for functions). A *partial set* on domain D is a function from D to $\mathcal{T}hree = \{\mathbf{t}, \mathbf{u}, \mathbf{f}\}$. A *partial* value of a predicate type $\tau' = (\bar{\tau} \to \mathbb{B})$ in domain D is a *partial set* with domain $\bar{\tau}^D$. $\mathcal{T}hree$ extends $\mathcal{T}wo$ and is equipped with two partial orders: the truth order $\leq$ is the least partial order satisfying $\mathbf{f} \leq \mathbf{u} \leq \mathbf{t}$, the precision order $\leq_p$ the least partial order satisfying $\mathbf{u} \leq_p \mathbf{f}, \mathbf{u} \leq_p \mathbf{t}$. The orders $\leq$ and $\leq_p$ on $\mathcal{T}hree$ are pointwise extended to partial sets. $\mathbf{u}$ is seen as an approximation of truth values, not as a truth value in its own right. A partial set that is maximally precise has range $\mathcal{T}wo$ and is called *exact*. A partial set $\mathcal{S}$ is seen as an approximation of any exact set S for which $\mathcal{S} \leq_p S$.

Example 2.3.1. The partial values for a predicate P typed as $\delta \to \mathbb{B}$ with domain $\delta = \{1, 2, 3\}$ are the functions from $\{1, 2, 3\}$ to $\{\mathbf{t}, \mathbf{u}, \mathbf{f}\}$. δ. The partial values for a second-order predicate P typed as $(\delta \to \mathbb{B}) \to \mathbb{B}$ with the same domain are the functions which maps any exact function from $\{1, 2, 3\}$ to $\{\mathbf{t}, \mathbf{f}\}$, to $\{\mathbf{t}, \mathbf{u}, \mathbf{f}\}$. ▲

(Partial) Interpretations. A partial Σ-interpretation $\mathcal{I}$ consists of a suitable domain $\tau^{\mathcal{I}}$ for every type τ in Σ (which is the set of partial sets on ${\tau_d}^{\mathcal{I}}$ in case τ is a predicate type with domain type τ_d), and for every symbol $\sigma \in \Sigma$ of type τ a value $\sigma^{\mathcal{I}} \in \tau^{\mathcal{I}}$. An exact Σ-interpretation is one that assigns exact values. The class of partial Σ-interpretations is denoted $\mathcal{I}nt(\Sigma)$; the class of exact Σ-interpretations is $Int(\Sigma)$.

The precision order $\leq_p$ and truth order $\leq$ are extended to partial interpretations in the standard way: $\mathcal{I} \leq_p \mathcal{I}'$ if $\mathcal{I}, \mathcal{I}'$ interpret the same vocabulary Σ, have

the same values for all types and non-predicate symbols, and $P^{\mathcal{I}} \leq_p P^{\mathcal{I}'}$ for every predicate symbol $P \in \Sigma$. Likewise for the truth order $\leq$. We use $\mathcal{I}$ to denote a partial interpretation (which may be exact) and I to denote an exact interpretation.

The restriction of a Σ-interpretation $\mathcal{I}$ to $\Sigma' \subseteq \Sigma$ is denoted as $\mathcal{I}|_{\Sigma'}$. If $\mathcal{I}$ is a partial Σ-interpretation, σ a symbol (that might not belong to Σ) and v a well-typed value for σ, then $\mathcal{I}[\sigma : v]$ is the $(\Sigma \cup \{\sigma\})$-interpretation identical to $\mathcal{I}$ except that v is the value of σ.

Given an interpretation $\mathcal{I}$ of at least the types of Σ, a *domain atom* of an n-ary predicate symbol $P \in \Sigma$ of type $\bar{\tau} \to \mathbb{B}$ in $\mathcal{I}$ is a pair $(P, \overline{d})$ where $\overline{d} \in \bar{\tau}^{\mathcal{I}}$. It is denoted as $P(\overline{d})$. If $\mathcal{I}$ interprets P, a domain atom $P(\overline{d})$ has a truth value $P(\overline{d})^{\mathcal{I}} = P^{\mathcal{I}}(\overline{d})$.

For any $v \in \mathcal{T}hree$ and set X of domain atoms of partial interpretation $\mathcal{I}$, we denote $\mathcal{I}[X : \mathbf{t}]$ the interpretation identical to $\mathcal{I}$ except that each $A \in X$ is true; similarly for $\mathcal{I}[X : \mathbf{u}], \mathcal{I}[X : \mathbf{f}]$. We may concatenate such notions and write $\mathcal{I}[X : \mathbf{u}][Y : \mathbf{f}]$, with the obvious meaning (first revising X, next revising Y).

Logics $\mathcal{L}^{\upsilon}$. A logic is specified as a pair $(\mathcal{L}, \upsilon)$ (denoted $\mathcal{L}^{\upsilon}$) such that $\mathcal{L}$ is a function mapping vocabularies Σ to sets $\mathcal{L}(\Sigma)$ of expressions over Σ, and υ is a two-valued or three-valued truth assignment. An expression φ of $\mathcal{L}(\Sigma)$ has free symbols in Σ; it may contain other symbols provided they are bound by some scoping construct in a subexpression of φ (e.g., a quantifier). If $\Sigma \subseteq \Sigma'$, then $\mathcal{L}(\Sigma) \subseteq \mathcal{L}(\Sigma')$.

A (three-valued) truth assignment υ maps tuples $(\varphi, \mathcal{I})$ where $\mathcal{I}$ interprets all free symbols of φ, to $\mathcal{T}hree$. This function satisfies the following properties:

1. if $\varphi \in \mathcal{L}(\Sigma)$, $\Sigma \subseteq \Sigma'$ and $\mathcal{I}$ is a Σ'-interpretation, then $\varphi^{\upsilon:\mathcal{I}} = \varphi^{\upsilon:\mathcal{I}|_{\Sigma}}$
2. *exactness*: $\varphi^{\upsilon:I} \in \mathcal{T}wo$ for every exact interpretation I
3. $\leq_p$*-monotonicity*: if $\mathcal{I} \leq_p \mathcal{I}'$ then $\varphi^{\upsilon:\mathcal{I}} \leq_p \varphi^{\upsilon:\mathcal{I}'}$.
4. If $\mathcal{I}$ is isomorphic to $\mathcal{I}'$ then $\varphi^{\upsilon:\mathcal{I}} = \varphi^{\upsilon:\mathcal{I}'}$

A two-valued truth assignment υ should adhere to the same restrictions, but is only defined for exact interpretations, so the $\leq_p$-monotonicity becomes trivial.

Definition 2.3.2. We use $\emptyset$ to denote the empty logic. This logic contains no expressions, and thus has a unique (empty) interpretation for the truth assignment function.

Definition 2.3.3. We say that two formulas φ_1 and φ_2 over Σ_1 and Σ_2 respectively are Σ-*equivalent*, with $\Sigma \subseteq (\Sigma_1 \cap \Sigma_2)$, if for any interpretation I over Σ, there exists an expansion I_1 to Σ_1 for which $\varphi^{\upsilon:I_1} = \mathbf{t}$ iff there exists an expansion I_2 of I to Σ_2 for which $\varphi^{\upsilon:I_2} = \mathbf{t}$. If in addition $\Sigma_1 = \Sigma_2 = \Sigma$, we call φ_1 and φ_2 equivalent; they have the same truth value in all Σ-interpretations.

Views. A three-valued semantics for $\mathcal{L}$ induces a two-valued one. In particular, υ induces a standard satisfaction relation: $I \models \varphi$ if $\varphi^{\upsilon:I} = \mathbf{t}$; in turn this defines the entailment relation: $\varphi \models \psi$ if $I \models \varphi$ implies $I \models \psi$. As common in Tarskian semantics, we view an exact interpretation as a mathematical abstraction of a states of affair and $\varphi^{\upsilon:I}$ as whether this state of affairs satisfies φ. Partial interpretations are not representations of states of affairs but approximations of these.

2.4 Well-founded and stable semantics for $\mathcal{L}$-rule sets

In this section, we show that for each logic $\mathcal{L}^{\upsilon}$ with a three-valued truth assignment υ, it is possible to define a rule logic under a well-founded and under a stable semantics. Let $\mathcal{L}^{\upsilon}$ be such a logic.

Definition 2.4.1. An $\mathcal{L}$-rule over Σ is an expression $\forall \overline{x}(P(\overline{x}) \leftarrow \varphi)$ with P a predicate symbol in Σ, $\overline{x}$ a tuple of "variable" symbols and $\varphi \in \mathcal{L}(\Sigma \cup \{\overline{x}\})$. An $\mathcal{L}$-rule set over Σ is a set of $\mathcal{L}$-rules over Σ. Rule sets will be denoted with Δ.

The set $\mathit{Def}(\Delta)$ is the set of predicate symbols $P \in \Sigma$ that occur in the head of a rule. $\mathit{Par}(\Delta)$ is the set of all other symbols that occur in Δ. Elements of $\mathit{Def}(\Delta)$ are called *defined* symbols, the other ones are called *parameters* of Δ. A rule set Δ is seen as a definition of its defined symbols in terms of its parameters. This motivates the following definition.

Definition 2.4.2. A context $\mathcal{O}$ of a $\mathcal{L}$-rule set Δ is a $\Sigma \setminus \mathit{Def}(\Delta)$-interpretation.

For a given context $\mathcal{O}$, the set $\{\mathcal{I} \mid \mathcal{I}|_{\mathit{Par}(\Delta)} = \mathcal{O}\}$ of partial Σ-interpretations expanding $\mathcal{O}$ is isomorphic to the set of partial sets of domain atoms of $\mathit{Def}(\Delta)$ in $\mathcal{O}$. Thus, given $\mathcal{O}$, a partial set of domain atoms specifies a unique partial interpretation $\mathcal{I}$ expanding $\mathcal{O}$ and vice versa.

We call a set of domain atoms a **t**-set, respectively **u**-set, **f**-set of partial interpretation $\mathcal{I}$ if its elements have truth value **t**, respectively **u**, **f** in $\mathcal{I}$.

Definition 2.4.3. A partial interpretation $\mathcal{I}$ is closed under Δ if for any domain atom $P(\overline{d})$ and rule $\forall\overline{x}(P(\overline{x}) \leftarrow \varphi) \in \Delta$, if $\varphi[\overline{d}]^{v:\mathcal{I}} = \mathbf{t}$ then $P(\overline{d})^{v:\mathcal{I}} = \mathbf{t}$.

Definition 2.4.4. An unfounded set of Δ in $\mathcal{I}$ is a $\mathbf{u}$-set U of defined domain atoms in $\mathcal{I}$, for which every atom $P(\overline{d}) \in U$ and rule $\forall\overline{x}(P(\overline{x}) \leftarrow \varphi) \in \Delta$, $\varphi[\overline{d}]^{v:\mathcal{I}[U:\mathbf{f}]} = \mathbf{f}$.

Definition 2.4.5. A partial interpretation $\mathcal{I}$ extending context $\mathcal{O}$ is a *partial stable interpretation* of Δ if

1. for each domain atom $P(\overline{d})$, $P(\overline{d})^{\mathcal{I}} = Max_{\leq}\{\varphi[\overline{d}]^{v:\mathcal{I}} \mid \forall\overline{x}(P(\overline{x}) \leftarrow \varphi) \in \Delta\}$;
2. (prudence) there exists no non-empty $\mathbf{t}$-set T and no (possibly empty) $\mathbf{u}$-set U of $\mathcal{I}$ such that $\mathcal{I}[T : \mathbf{u}][U : \mathbf{f}]$ is closed under Δ;
3. (braveness) the only unfounded set of Δ in $\mathcal{I}$ is $\emptyset$.

Definition 2.4.6. We call a partial interpretation $\mathcal{I}$ a well-founded interpretation of Δ if $\mathcal{I}$ is the $\leq_p$-least partial stable model $\mathcal{I}'$ of Δ such that $\mathcal{I}'|_{Par(\Delta)} = \mathcal{I}|_{Par(\Delta)}$.

Definition 2.4.7. We call an (exact) interpretation I a stable interpretation of Δ if I is an exact partial stable model of Δ.

Given that a stable I has only the empty $\mathbf{u}$-set, conditions (2) and (3) simplify to that there is no non-empty $\mathbf{t}$-set T of $\mathcal{I}$ such that $\mathcal{I}[T : \mathbf{u}]$ is closed under Δ.

Proposition 2.4.8. *Let φ, φ' be equivalent under v (same truth value in all partial interpretations). Then substituting φ for φ' in the body of a rule of Δ preserves the class of partial stable (hence, well-founded and stable) interpretations.*

Proof. This is trivial, since the conditions of partial stable interpretation are defined in terms v, which cannot distinguish φ from φ'. □

Two logics. Using the above two concepts we define two rule logics. Expressions in both logics are the same: finite sets of rules.

Definition 2.4.9. For logic $\mathcal{L}^v$, we define logic $R(\mathcal{L}^v)^w$ where $R(\mathcal{L}^v)(\Sigma)$ is the collection of finite rule sets over Σ and w the two-valued truth assignment defined as $\Delta^{w:I} = \mathbf{t}$ if I is an exact well-founded interpretation of Δ and $\Delta^{w:I} = \mathbf{f}$ otherwise.

Definition 2.4.10. For logic $\mathcal{L}^v$, we define the logic $R(\mathcal{L}^v)^{st}$ where $R(\mathcal{L}^v)(\Sigma)$ is as above and *st* is the two-valued truth assignment defined as $\Delta^{st:I} = \mathbf{t}$ if I is an exact stable interpretation of Δ and $\Delta^{st:I} = \mathbf{f}$ otherwise.

For the logic FO^k, with FO first order logic and k the 3-valued Kleene truth assignment [82], the rule formalism $R(FO^k)^w$ corresponds to the (formal) definitions in the logic FO(ID) [38, 42] while the formalism $R(FO^k)^{st}$ corresponds to the rule formalism in the logic ASP-FO [39].

In [44], the relation between the main forms of (informal) *definitions* found in mathematical text, and rule sets in $R(FO^k)^w$ was analyzed. Not all rule sets of $R(FO)$ express sensible (informal) definitions, but for those that do, the well-founded interpretations are exact and correctly specify the defined sets. Therefore, a rule set Δ was called a *paradox-free* or *total definition* in an exact context $\mathcal{O}$ if its well-founded interpretation expanding $\mathcal{O}$ is exact. For paradox-free rule sets, w and *st* coincide. Important classes of rule sets are always paradox-free: non-recursive, monotone inductive rule sets, and rule sets by ordered or iterated induction over some well-founded induction order. They are defined and proven to be paradox-free in [44].

2.5 Compositional framework for building logics with definitions

This section effectively defines an infinite collection of logics. We define compositional constructs which add new expressions, such as definitions, to an existing logic. By iterating such extension steps, these constructs can be nested.

2.5.1 Approximating boolean functions

We frequently need to extend a boolean function defined on a domain X of exact values (e.g., $\mathcal{T}wo$, exact sets, exact interpretations, or tuples including these) to the domain $\mathcal{X}, \leq_p$ of partial values. Examples are the boolean functions associated with connectives $\neg, \wedge, \ldots$, or the truth assigments w and *st* of $R(\mathcal{L}^v)$ defined on $Int(\Sigma)$. Given such a function $F : X \to \mathcal{T}wo$, we search for an approximation $\mathcal{F} : \mathcal{X} \to \mathcal{T}hree$ such that:

- $\leq_p$-monotone: if $\mathbf{x} \leq_p \mathbf{y} \in \mathcal{X}$ then $\mathcal{F}(\mathbf{x}) \leq_p \mathcal{F}(\mathbf{y})$;
- exact and extending: for $x \in X$, $\mathcal{F}(x) = F(x)$.

Definition 2.5.1. We define the ultimate approximation $\widetilde{F} : \mathcal{X} \to \mathcal{T}hree$ of F by defining $\widetilde{F}(\mathbf{x}) = \text{glb}_{\leq_p}\{F(x) \mid \mathbf{x} \leq_p x\} \in \mathcal{T}hree$.

Proposition 2.5.2. *$\widetilde{F}$ is the most precise $\leq_p$-monotone exact extension of F.*

Proof. $\leq_p$-monotonicity follows from the transitivity of $\leq_p$. Exactness, from the fact that elements of X are maximally precise. That $\widetilde{F}$ is the most precise approximation of F is clear as well. □

Note that this ultimate approximation can be used on a wide range of functions, going as specific as boolean connectives or going as broad as a full truth assignment function. In the next paragraphs we show how this ultimate approximator behaves for a range of possible applications.

Boolean Connectives For a standard connective $c \in \{\wedge, \vee, \neg, \Rightarrow, \Leftrightarrow\}$ with corresponding boolean function $\mathbf{c} : \mathcal{T}wo^n \to \mathcal{T}wo$, the function $\widetilde{\mathbf{c}} : \mathcal{T}hree^n \to \mathcal{T}hree$ is the three-valued truth function used in the Kleene truth assignment k.

(Generalized) Quantifiers The semantics of quantifiers $\forall, \exists$ and generalized quantifiers such as aggregates are given by functions on sets (or tuples including sets). E.g., for quantification over domain D these are the boolean functions $\forall_D, \exists_D$ defined $\forall_D(S) = \mathbf{t}$ iff $D \subseteq S$, and $\exists_D(S) = \mathbf{t}$ iff $D \cap S \neq \emptyset$. Two commonly used numerical aggregate functions are cardinality $\#$ and *sum* (the latter mapping (finite) sets S of tuples $\bar{d}$ to $\sum_{\bar{d} \in S} d_1$). For every numerical aggregate function Agg and boolean operator $\sim \in \{=, <, >\}$ on numbers, the boolean function $Agg_\sim$ maps tuples (S, n) to $\mathbf{t}$ iff $Agg(S) \sim n$.

Example 2.5.3. The ultimate approximation of the quantifier $\forall_D$ maps partial sets to $\mathcal{T}hree$. $\widetilde{\forall_D}(\mathcal{S}) = \mathbf{t}$ iff $\forall S \geq_p \mathcal{S} : D \subseteq S$. $\widetilde{\forall_D}(\mathcal{S}) = \mathbf{f}$ iff $\forall S \geq_p \mathcal{S} : D \supsetneq S$. $\widetilde{\forall_D}(\mathcal{S}) = \mathbf{u}$ iff $\exists S \geq_p \mathcal{S} : D \subseteq S \wedge \exists S' \geq_p \mathcal{S} : D \supsetneq S$. In other words, the quantification is true if in all more precise cases the quantification is true. The quantification is false if in all more precise cases the quantification is false. The quantification is unknown if there is a more precise case in which the quantification is true and one in which it is false. ▲

Higher Order Functions For all these higher order boolean functions F, $\widetilde{F}$ is the most precise approximation on three-valued sets. The three-valued aggregate functions $\widetilde{Agg_\sim}$ were introduced originally in [105] to define stable and well-founded semantics for aggregate logic programs. The functions $\widetilde{\forall}_D, \widetilde{\exists}_D$ are used in the Kleene truth assignment k: let D be $\tau(x)^{\mathcal{I}}$, and $\mathcal{S} = \{(d, \varphi^{k:\mathcal{I}[x:d]}) \mid d \in$

$D\}$, i.e., the three valued set mapping domain elements $d \in D$ to $\varphi^{k:\mathcal{I}[x:d]}$), one defines $(\forall x\ \varphi)^{k:\mathcal{I}} = \widetilde{\forall}_D(\mathcal{S}) = Min_{\leq}\{\mathcal{S}(d) \mid d \in D\} = Min_{\leq}\{\varphi^{k:\mathcal{I}[x:d]}) \mid d \in D\}$.

Truth Assignments For any two-valued truth assigment v on $\mathcal{L}$, $\widetilde{v}$ is a sound three-valued truth assignment. In case of FO and its truth assignment v, $\widetilde{v}$ was introduced in [123] where it was called the supervaluation s. s is not truth functional, for if $p^{\mathcal{I}} = q^{\mathcal{I}} = \mathbf{u}$, then $(p \vee \neg p)^{s:\mathcal{I}} = \mathbf{t} \neq (p \vee q)^{s:\mathcal{I}} = \mathbf{u}$ while the components of the two disjunctions have the same supervaluation. A truth-functional definition of a three-valued truth assignment is obtained by using the ultimate approximations of the boolean functions associated to connectives and quantifiers. This yields exactly the Kleene truth assigment k. It is $\leq_p$-monotone, exact and extending, and strictly less precise than s as can be seen from $(p \vee \neg p)^{k:\mathcal{I}} = \mathbf{u}$: the supervaluation "sees" the logical connection between p and $\neg p$ in this tautology while k does not.

Other applications serve to extend, for arbitrary logic $\mathcal{L}^v$, the two-valued wellfounded and stable truth assignments w and st on $R(\mathcal{L}^v)$ to three-valued extensions $\widetilde{w}, \widetilde{st}$.

Proposition 2.5.4. *It holds that* $\Delta^{\widetilde{w}:\mathcal{I}} = \mathbf{t}$ *(respectively* $\mathbf{f}$*) if every (respectively, no) two-valued structure* $I \geq_p \mathcal{I}$ *is a well-founded interpretation of* Δ.

Proof. We prove this proposition for the value $\mathbf{t}$, the proof for $\mathbf{f}$ is analogous. From Definition 2.5.1 we know that $\widetilde{w}(\mathcal{I}) = \mathrm{glb}_{\leq_p}\{w(I) \mid \mathcal{I} \leq_p I\}$. If all $I \geq_p \mathcal{I}$ are well-founded interpretations of Δ, then it follows from Definition 2.4.9 that all $w(I)$ will be $\mathbf{t}$. This means that $\widetilde{w}(\mathcal{I})$ will be the greatest lower bound of the singleton $\mathbf{t}$, which is $\mathbf{t}$. □

2.5.2 Composing logics by combining logic constucts

A standard way of defining the syntax of a logic is through a set of often inductive syntactical rules, typically described in Backus Naur Form (BNF). The truth assignment v is then defined by recursion over the structure of the expressions. Below, we identify a language construct C with a pair of a syntactical and a semantical rule. The rules below construct, for a language construct C, a new logic $C(\mathcal{L}^v)^{v'}$ with expressions obtained by applying C on subexpressions of $\mathcal{L}^v$ and v' a truth assignment for $C(\mathcal{L})$. Afterwards, complex logics with multiple and nested language constructs can be built by iterating these construction steps.

- Atom^1 and Atom^2: for first order predicates p and second order ones P respectively. $\overline{t}$ is a tuple of terms, $\overline{x}$ of variables.

$$Atom^1(\mathcal{L}^v)(\Sigma) ::= \{p(\overline{t}) \mid p \in \Sigma, \overline{t} \text{ a tuple of terms}\}$$

$$\text{where } p(\overline{t})^{v':\mathcal{I}} = p^{\mathcal{I}}(t_1^{\mathcal{I}}, \cdots, t_n^{\mathcal{I}})$$

$$Atom^2(\mathcal{L}^v)(\Sigma) ::= \{P(\overline{t}) \mid P \in \Sigma, \overline{t} \text{ a tuple of terms}\}$$

$$\text{where } P(\overline{x})^{v':\mathcal{I}} = P^{\mathcal{I}}(x_1^{\mathcal{I}}, \cdots, x_n^{\mathcal{I}})$$

- N-ary connectives $c \in \{\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow\}$,

$$c(\mathcal{L}^v)(\Sigma) ::= \{c(\alpha_1, \ldots, \alpha_n) \mid \alpha_i \in \mathcal{L}(\Sigma)\}$$

$$\text{where } c(\alpha_1, \ldots, \alpha_n)^{v':\mathcal{I}} = \widetilde{\mathbf{c}}({\alpha_1}^{v:\mathcal{I}}, \cdots, {\alpha_n}^{v:\mathcal{I}})$$

- Generalized quantifiers $C \in \{\forall, \exists, Agg_\sim\}$ where $\sim \in \{=, <, >\}$. Below, $C\langle \overline{x}, \alpha, z\rangle$ denotes the syntactic expression, e.g., $\forall\langle x, \alpha\rangle$ is $\forall x\ \alpha$; $Agg_\sim\langle \overline{x}, \alpha, z\rangle$ is $Agg\{\overline{x} : \alpha\}{\sim}z$.

$$C(\mathcal{L}^v)(\Sigma) ::= \{C\langle \overline{x}, \alpha, z\rangle \mid \alpha \in \mathcal{L}(\Sigma), \overline{x} \text{ a tuple of identifiers, } z \text{ a term}\}$$

$$\text{where } (C\langle \overline{x}, \alpha, z\rangle)^{v':\mathcal{I}} = \widetilde{\mathcal{C}}(\{(\overline{d}, \alpha^{v:\mathcal{I}[\overline{x}:\overline{d}]}) | \overline{d} \in \tau(\overline{x})^{\mathcal{I}}\}, z^{\mathcal{I}})$$

- Definitions as rule sets (R_w) (similarly, one could define R_{st}):

$$R_w(\mathcal{L}^v)(\Sigma) ::= R(\mathcal{L})(\Sigma) \text{ where } \Delta^{v':\mathcal{I}} = \Delta^{\widetilde{w}:\mathcal{I}}$$

  where w is the well-founded assignment of $R(\mathcal{L}^v)$ as defined in Section 4.

Example 2.5.5. For any logic $\mathcal{L}^v$, $\wedge(\mathcal{L}^v)$ is a new logic consisting of binary conjunctions of two expressions in the original language. Note that this new logic does not contain nested conjunctions and the original expressions from $\mathcal{L}^v$ are no longer in $\wedge(\mathcal{L}^v)$. ▲

Example 2.5.6. For any logic $\mathcal{L}^v$, $Atom^1(\mathcal{L}^v)$ is the same logic consisting of first order atoms. This logic is the same, independently of the original logic $\mathcal{L}^v$. ▲

Proposition 2.5.7. *The union of two logics $\mathcal{L}^v$ and ${\mathcal{L}'}^{v'}$ is a valid new logic. The expressions of $\mathcal{L}^v \cup {\mathcal{L}'}^{v'}$ are the union of the expressions of $\mathcal{L}^v$ and ${\mathcal{L}'}^{v'}$. The semantics of all the expressions in $\mathcal{L}^v \cup {\mathcal{L}'}^{v'}$ are equal to the semantics in their original logic.*

Building logics. Using the above rules of language constructs, an (infinite) class of logics with three-valued semantics can be built. Moreover, every combination of the above rules gives rise to a valid three-valued truth assignment.

Definition 2.5.8. For every set of language constructs $\{c_1, \ldots, c_n\}$, the logic $\{c_1, \ldots, c_n\}^*$ is defined as $\bigcup_{k \in \mathbb{N}} \mathcal{L}_k$ where:

$$\mathcal{L}_0 ::= \emptyset$$

$$\mathcal{L}_k ::= \bigcup_{i \in 1 \ldots n} c_i(\mathcal{L}_{k-1})$$

Example 2.5.9. The logic $\{Atom^1, \neg\}^*$ is a propositional logic. For this logic $\mathcal{L}_1$ corresponds all atoms without any negations. $\mathcal{L}_2$ corresponds to all atoms and their negations. $\mathcal{L}_3$ corresponds to all atoms with one or two negations, and so on. ▲

Proposition 2.5.10. *Every (sub)set of the above language constructs S induces a logic S^* with a proper three-valued truth assignment (i.e., it is $\leq_p$-monotone, exact and extending).*

Proof. Taking the union of logics, results in a valid logic (cf. Propostion 2.5.7). $S*$ is a big union of logics, so we only need to prove that the individual logics in this union have a proper three-valued truth assignment. To do this, we need to proof that all the above language constructs map a valid logic to a new logic with a proper three-valued truth assignment. The three-valued truth assignments of the logics by these language constructs are defined through the ultimate approximations. From Proposition 2.5.2, we know that these truth assignments are a proper three-valued truth assignment. □

For example, given a logic $\mathcal{L}^v$, we define $\mathcal{L}'^{v'} = R(\mathcal{L}^v)^{\widetilde{w}}$ by one application of R_w on $\mathcal{L}^v$. By iterating R_w, logics $(R(\mathcal{L}^v)^{\widetilde{w}})^n$ with nested definitions are built. Every BNF in terms of the above language constructs now implicitly defines a three-valued logic. The definition of first order logic FO^k with k the standard three-valued Kleene truth assignment, can be descibed in BNF or more compactly as $\{Atom^1, \wedge, \vee, \neg, \forall, \exists\}^*$.

The logic FO(ID) defined in [42] is the union of logics FO^k and $R(FO^k)^{\widetilde{w}}$. A further extension is the new logic $FO(ID^*) = \{Atom^1, \wedge, \vee, \neg, \forall, \exists, R_w\}^*$ which has definitions nested in formulas and definition rule bodies. A logic in which templates can easily be embedded is $SO(ID^*) = \{Atom^1, Atom^2, \wedge, \vee, \neg, \forall, \exists, R_w\}^*$. It is a second order extension of FO(ID) which allows for nesting of definitions in rule bodies.

If we combine this technique with the one of the previous section, we find that a suitable logic $\mathcal{L}$ induces a logic $\mathcal{L} \cup R_w(\mathcal{L}^v)$. An example of such a logic that appeared in the literature is the logic FO(ID) [38, 42]. Here $\mathcal{L}$ is classical first order logic FO and $FO(ID) = FO \cup D(FO)$. Also the logic $FO \cup R_{st}(FO)$ was introduced in [39] as the logic ASP-FO.

2.6 Templates

Now that we built SO(ID^*), we proceed by introducing compositionality in our logic specifications through templates. We envision a library of application independent templates in the form of second order definitions that encapsulate prevalent patterns and concepts and that can be used as building blocks to compose logic specifications. Below, we formally define the concepts and show that non-recursive templates do not increase the computational complexity of FO(ID^*) and can be eliminated by a rewriting process.

2.6.1 Definition and usage

We assume the existence of a set of template symbols. A *template atom* is an atom $s(\overline{x})$, where s is a template symbol. A template is a context-agnostic second order definition of template symbols. As such it should define and contain only domain independent symbols: interpreted symbols and template symbols. A template might define a template symbol in terms of other template symbols and interpreted predicates, but not in terms of user-defined symbols.

Definition 2.6.1. The template vocabulary Σ_{Temp} is the vocabulary consisting of all interpreted symbols (such as arithmetic symbols) and all template symbols.

Definition 2.6.2. A template is a second order definition Δ over Σ_{Temp} such that $Def(\Delta)$ consists of template symbols.

Thus the set of parameters $Par(\Delta)$ of a template consists only of interpreted symbols and template symbols.

The concepts used in Listing 2.1 are now fully defined. Another common example is the template `tc` expressing that Q/2 is the transitive closure of P/2, as shown in Listing 2.2. Note that this example cannot be written without a definition in the body of the template, so this further motivates our choice to allow definitions in the bodies of other definitions in our recursive construction of the logic $SO(ID^*)$.

Listing 2.2: Q is the transitive closure of P

```
{∀P ∀Q: tc(P,Q) ← {∀x ∀y: Q(x,y) ← P(x,y) ∨(∃ z: Q(x,z)∧Q(z,y))}.
}
```

Listing 2.3: P is the range of integers from a to b

```
{∀P ∀a ∀b: range(P, a, b) ←
  {P(a) ← a < b.
   ∀ x: P(x) ← a < b ∧ (∃ Q : range(Q,a+1,b) ∧ Q(x)).
  }
}
```

Another notable aspect of this approach to templates is that recursive templates are well-defined. This enables us to write recursive templates, for example to define a range as shown in Listing 2.3

It is possible to rewrite Listing 2.3 into a non-recursive template. Listing 2.4 contains an example which is not rewritable in such a way. This template defining *win* and *lose* by simultaneous definition, is a monotone second order definition and has a two-valued well-founded model.

Proposition 2.6.3. *Assuming the complexity classes PH and PSPACE are not equal. The recursive second order definition of winning positions in a two-player game as shown in Listing 2.4 cannot be rewritten into a non-recursive version.*

Proof. Suppose we have a non-recursive version of the same definition. Suppose we specialise this definition for (a generalized version of) the game Othello. This process instantiates the definition with a particular argument for Move and IsWon. This specialised definition determines the winning and losing positions in Othello. It is well-known that determining these positions is PSPACE-complete[75].

Listing 2.4: cur is a winning position in a two-player game

```
{∀cur ∀Move ∀IsWon: win(cur, Move, IsWon) ← IsWon(cur) ∨
                      ∃ nxt : Move(cur,nxt) ∧ lose(nxt,Move,IsWon).
 ∀cur ∀Move ∀IsWon: lose(cur,Move, IsWon) ← ¬IsWon(cur) ∧
                      ∀ nxt : Move(cur,nxt) ⇒ win(nxt,Move,IsWon).
}
```

Deciding the truthness of a second order logic statement is in PH[118]. A non-recursively defined second order predicate can be expressed as its completion, so deciding if a tuple belongs to a non-recursively defined second order predicate is in PH. So now we see that the non-recursive definition can solve a PSPACE-complete problem and is in PH. As PH $\subset$ PSPACE, this contradicts our assumption that PH and PSPACE are not equal. So this means that there cannot be a non-recursive version of the definition. □

Definition 2.6.4. A template library L is a finite set of templates satisfying (1) every template is paradox-free; (2) every template symbol is defined in exactly one template; (3) the set of templates is hierarchically stratified: there is a strict order $<$ on template symbols such that for each $\Delta \in L$, if $P \in \mathit{Def}(\Delta), Q \in \mathit{Par}(\Delta)$ then $Q < P$.

Proposition 2.6.5. *For a template library L, each interpretation I not interpreting symbols of Σ_{Temp} has a unique two-valued expansion I' to Σ_{Temp} that satisfies L.*

Proof. Multiple hierarchically stratified definitions under the well-founded semantics can be merged into one larger definition without any change of semantics. From this it follows that a template library can be represented as a single well-founded definition. As every individual template is paradox-free, this large definition is also paradox-free. From this it follows that there is a unique well-founded model for the template library. □

2.6.2 The Σ_{Temp} vocabulary restriction

We want a template to express a concept which can be understood without the need for a context. Therefore it is reasonable to require that templates have no implicit dependence on another domain, or vocabulary symbols. Requiring that templates explicitly state their dependencies ensures that templates are 'drop-in' building blocks. This restriction might seem too stringent, but we can show that many template definitions for which it does not hold, can be rewritten as an equivalent one for which it holds.

Let Δ be a definition of second order predicates with $\mathit{Def}(\Delta) \cap \Sigma_{Temp} = \emptyset$, and $\overline{o}$ the tuple of all free (user-defined) symbols of $\mathit{Par}(\Delta) \setminus \Sigma_{Temp}$ (arranged in some arbitrary order). For such definitions, we define a templified version. For any rule or formula Ψ, we define $\Psi^{\overline{o}}$ to be Ψ except that every atom $P(\overline{t})$ in Ψ with $P \in \mathit{Def}(\Delta)$ is replaced by $P'(\overline{t}, \overline{o})$, with P' a new symbol extending P with new arguments corresponding to $\overline{o}$.

Listing 2.5: A definition and its templified version

(a) A second order definition

```
{∀x union(x) ← P(x).
 ∀x union(x) ← Q(x).
}
```

(b) The templified version

```
{∀x union(x,P,Q) ← P(x).
 ∀x union(x,P,Q) ← Q(x).
}
```

We say that a structure I corresponds to I' if I, I' interpret the free symbols of Δ, respectively those of Δ_{Temp}, such that they are identical on shared symbols and for each $P \in Def(\Delta)$, $P^I = \left\{\overline{d} | (\overline{d}, \overline{o}^I) \in P'^{I'}\right\}$. Note that for each I' and each value $\overline{d}_o$ for $\overline{o}$ in the domain of I', there is a unique interpretation I with $\overline{o}^I = \overline{d}_o$ that corresponds to I'.

Definition 2.6.6. We define the *templified definition* Δ_{Temp} of Δ as the definition $\{\forall\overline{o}(\Psi^{\overline{o}}) \mid \Psi \in \Delta\}$ and we define $\Sigma'_{Temp} = \Sigma_{Temp} \cup \{P' \mid P \in Def(\Delta)\}$.

We assume that $\overline{o}$ consists only of first order predicate symbols. Under this condition, the templified definition Δ_{Temp} is a template over Σ'_{Temp}.

Example 2.6.7. Listing 2.5(a) contains a definition of `union` as the union of the predicates `P` and `Q`. This definition contains two parameters: `P` and `Q`. In the templified version seen in Listing 2.5(b), these parameters are made explicit and added to the arguments of the `union` relation. ▲

Proposition 2.6.8. *Let I be a well-founded model of Δ and I' a well-founded model of Δ_{Temp} such that I and I' are identical on Σ_{Temp}. Then it holds that*

$$P^I = \left\{\overline{d} | (\overline{d}, \overline{o}^I) \in P'^{I'}\right\}$$

Stated differently, I corresponds to I'. The templified definition captures the original one, and hence, each theory can be rewritten in terms of the new templified defined symbols.

Proof. Assume that I corresponds to I'. It is easy to prove, by induction on the formula structure, that for any formula φ in the vocabulary of Δ, it holds that $\varphi^I = (\varphi^{\overline{o}})^{I'[\overline{o}:\overline{o}^I]}$. We call this the independency property since it shows that $(\varphi^{\overline{o}})^{I'[\overline{o}:\overline{o}^I]}$ is influenced by only a small part of the interpretation of P', namely the values of domain atoms $P'(\overline{d}, \overline{o}^I)$.

The key property to prove is that I' is a partial stable interpretation of Δ_{Temp} iff for each value $\overline{d}_o$ for $\overline{o}$, the unique I that corresponds to I' such that $\overline{o}^I = \overline{d}_o$

is a partial stable interpretation of Δ. Intuitively, a partial stable interpretation of Δ_{Temp} is a kind of union of partial stable interpretations of Δ, one for each assignment of values to $\overline{o}$.

We prove this property only in one direction. The other direction is similar. Assume that I' is a partial stable interpretation of Δ_{Temp} satisfying the three conditions of Definition 2.4.5. We need to show for every I that corresponds to I', that I is a partial stable interpretation of Δ. Condition 1), that $P(\overline{d})^I = Max_{\leq}\{\varphi[\overline{d}]^I \mid \forall\overline{x}(P(\overline{x}) \leftarrow \varphi[\overline{x}]) \in \Delta\}$ follows from the fact that $P'(\overline{d},\overline{o}^I)^{I'}$ satisfies the corresponding equation for Δ_{Temp}, that $P(\overline{d})^I = P'(\overline{d},\overline{o}^I)^{I'}$, and that for each rule body φ for P, $\varphi[\overline{d}]^I = (\varphi^{\overline{o}})^{I'[\overline{o}:\overline{o}^I]}$ (by the independency property). The condition 2) follows from the fact that when $T' = \{P'(\overline{d},\overline{o}^I) \mid P(\overline{d}) \in T\}$, and $U' = \{P'(\overline{d},\overline{o}^I) \mid P(\overline{d}) \in U\}$, then T is a $\mathbf{t}$-set and U a $\mathbf{u}$-set of I such that $I[T:\mathbf{u}][U:\mathbf{t}]$ is closed under Δ iff T' is a $\mathbf{t}$-set and U' a $\mathbf{u}$-set of I' such that $I'[T':\mathbf{u}][U':\mathbf{t}]$ is closed under Δ_{Temp}. This follows from the independency property. Condition 3) is proven similarly.

It is easy to see that this property entails the proposition, since intuitively, it entails that a well-founded model I' of Δ_{Temp}, which is the $glb_{\leq_p}$ of all partial stable interpretations of Δ_{Temp} with the same context as I', contains for each value $\overline{d}_o$ for $\overline{o}$ the $glb_{\leq_p}$ of the partial stable interpretations I of Δ in the context with $\overline{o}^I = \overline{d}_o$. □

2.6.3 Simple Templates

Extending a logic with arbitrary (recursive) templates may easily increase the descriptive complexity of the logic. Below, we develop a simple but useful template formalism for FO(ID) that does not have this effect. In addition, we show that libraries of simple templates can be compiled away using them as macros.

In Figure 2.1 we define sublanguages FO(ID^*), ESO(ID^*) and ASO(ID^*) of SO(ID^*) (by mutual recursion) consisting of atoms, negations, conjunctions, quantification, definitions and the let-construct. This last construct represents a second order quantification, where the quantified symbol(s) S are defined in an accompanying paradox-free definition Δ. Definitions of second order symbols in ESO(ID^*) and ASO(ID^*) contain only (possibly nested) first order definitions. Since model checking of (nested) first order definitions is polynomial, the descriptive complexity of FO(ID^*) is P, of ESO(ID^*) is NP and of ASO(ID^*) is co-NP.

Definition 2.6.9. A *simple template* is a template of the form $\{\forall\overline{x}(P(\overline{x}) \leftarrow \varphi_P[\overline{x}])\}$ with $P(\overline{x}) \in Atom^2$ and $\varphi_P \in FO(ID^*)$.

$FO(ID^*)\ \varphi ::=$

$|s(\bar{t})(\in Atom^1)$

$|\neg\varphi$

$|\varphi \wedge \varphi$

$|\exists_{FO}\ s : \varphi$

$|\text{let } \{s(\bar{t}) \leftarrow \varphi\} \text{ in } \varphi$

$|\{s(\bar{t}) \leftarrow \varphi\}$

(a) FO(ID^*)

$ESO(ID^*)\ \epsilon ::=$

$|S(\bar{t})(\in Atom^2)$

$|\neg\alpha$

$|\epsilon \wedge \epsilon$

$|\exists_{FO}\ s : \epsilon$

$|\text{let } \{s(\bar{t}) \leftarrow \varphi\} \text{ in } \epsilon$

$|\{s(\bar{t}) \leftarrow \varphi\}$

$|\exists_{SO}\ s : \epsilon$

(b) ESO(ID^*)

$ASO(ID^*)\ \alpha ::=$

$|S(\bar{t})(\in Atom^2)$

$|\neg\epsilon$

$|\alpha \wedge \alpha$

$|\exists_{FO}\ s : \alpha$

$|\text{let } \{s(\bar{t}) \leftarrow \varphi\} \text{ in } \alpha$

$|\{s(\bar{t}) \leftarrow \varphi\}$

$|\forall_{SO}\ s : \alpha$

(c) ASO(ID^*)

Figure 2.1: The FO(ID^*), ESO(ID^*) and ASO(ID^*) subformalisms of SO(ID^*)

A simple template defines one symbol and contains one rule with an FO(ID^*) body. Let L be a template library over Σ_{Temp} consisting of non-recursive simple templates. Such a library is equivalent to the conjunction of the completion of its definitions $\forall\overline{x}(P(\overline{x}) \Leftrightarrow \varphi_P[\overline{x}])$. We want to show that while using such libraries increases convenience, reuse, modularity, and it does neither increase complexity, nor expressivity. Also, such libraries can be used in the common way, as macros.

Theorem 2.6.10. *Let L be a non-recursive simple template library. For $\Sigma \cap \Sigma_{Temp} = \emptyset$, let φ be an ESO(ID^*) formula over $\Sigma \cup \Sigma_{Temp}$ that does not contain definitions of template symbols. There exists a polynomially larger ESO(ID^*) formula φ_1 over Σ that is Σ-equivalent to $\{\varphi\} \cup L$. There exists a polynomially larger FO(ID^*) formula φ_2 over an extension Σ_1 of Σ that is Σ-equivalent to $\{\varphi\} \cup L$.*

Proof. The formula φ_1 is obtained by treating L as a set of macros. We iteratively substitute template atoms $P(\bar{t})$ in φ by $\varphi_P[\bar{t}]$. This process is equivalence preserving. It terminates due to the stratification condition on L, and the limit is a polynomially larger ESO(ID^*) formula φ_1 in the size of φ (exponential in $\#(L)$) that is Σ-equivalent to $\{\varphi\} \cup L$.

To obtain φ_2, we apply the well-known transformation of moving existential quantifiers to the front and skolemising them. Second order quantifiers can be

switched with first order ones using:

$$\forall_{FO}\, x : \exists_{SO} P : \varphi \Leftrightarrow \exists_{SO} P' : \forall_{FO}\, x : \varphi[P(\bar{t}) \backslash P'(\bar{t}, x)]$$

This process preserves Σ-equivalence. As only a polynomial number of steps are needed to transform the formula into this desired state, the size of the resulting formula is polynomially larger. □

Previous results in [18] indicated that the introduction of simple, stratified templates does not introduce a significant performance hit. The above theorem recovers these efficiency results.

2.7 Conclusion

In this chapter we developed a new way to define language constructs for logic. New language constructs must combine a syntactical rule with a three-valued semantic evaluation. This three-valued semantic evaluation is subject to certain restrictions. Language constructs can then be arbitrarily combined to compose a logic. In particular, we construct $SO(ID^*)$: a second order language with inductive definitions.

Using this language, it is easy to define templates as second order definitions. We conclude our chapter with a rewriting scheme to show that, given some restrictions, templates do not increase the descriptive complexity of the host language. In the next chapter we will generalize our way of defining language constructs to allow functions and provide a more comprehensive type system.

3

A Compositional Higher-Order Logic with Definitions

A first version of the work in this chapter was published as a technical communications paper at the International Conference on Logic Programming 2016 in New York City [34].

3.1 Introduction

Expressive knowledge representation languages consist of many different language constructs. New KR languages are often built by adding new (possibly nestable) language constructs to existing logics. Principled compositional methods are desired that allow to construct logics from language constructs, or incrementally extend an existing logic with a new construct, while preserving the meaning of the remaining language constructs. This is known as Frege's compositionality principle.

In classical monotone logics, it is common practice to extend a logic with new language constructs or connectives by specifying an additional pair of a syntactical and semantical rule. E.g., we can add a cardinality construct to classical first order logic (with finite structures) by defining:

- syntactical rule: $\#(\{x : \varphi\})$ is a (numerical) term if x is a variable and φ a formula;
- semantical rule: $(\#(\{x : \varphi\}))^I = \#(\{d \in dom(I) \mid I[x : d] \models \varphi\})$, the cardinality of the set of domain elements that correspond to the set expression. Here, $dom(I)$ is the domain of I.

In the context of nonmonotonic languages such as logic programming and extensions such as answer set programming [89, 87, 98] and the logic FO(ID) (classical logic with inductive definitions) [42], the situation is considerably more complex. For example, adding aggregates to these logics required, and still requires a serious effort [81, 124, 55, 117, 105, 52, 60] and resulted in a great diversity of logics.

In this chapter, we propose a compositional principle to build logics, and apply it to build a logic $\mathbb{L}$ integrating typed higher order lambda calculus with definitions represented as rule sets under well-founded semantics. The two main contributions of this chapter are:

- It introduces a compositional principle to build and integrate logics and puts it to the test by building an expressive logic including rule sets, with aggregates, lambda expressions, higher order rules, rule sets to express definitions by monotone, well-founded and iterated induction, definitions nested in rules, ... The semantical basis is the concept of *infon* which provides a semantical abstraction of the meaning of expressions and is related to intensional objects in intensional logics [71].
- The logic itself brings together the logics of logic programming and descendants such as answer set programming and FO(ID), and the logic of typed lambda calculus which has become the foundational framework for formal specification languages and functional programming. We illustrate the application of the resulting logic to build simple and elegant theories that express complex knowledge.

3.2 Related Work

3.2.1 Logics

Chapter 2 introduced a simpler version of the framework from the current chapter, using informal notions. There, the framework was used to construct a logic permitting inductive definitions within the body of other inductive definitions. In that logic, *templates* are (possibly inductive) second order definitions that allow nesting inductive definitions; this nesting is required to build, for instance, templates defining one predicate parameter as the transitive closure of another parameter. In this chapter, we present the framework with a more formal basis, using the concept of *infons* as the mathematical object corresponding to the semantics of a language construct, and identify the notion of *Frege's principle of compositionality* as the underlying goal of the framework.

This chapter explicitly allows the construction of *higher-order* logics. In the context of meta-programming [2], some logics with higher-order syntax already exist. One such example is HiLog [21], which combines a higher-order syntax with first-order semantics. HiLog's main motivation for this is to introduce a useful degree of higher order while maintaining decidability of the deduction inference. Another example is λprolog [95], which extends Prolog with (among others) higher-order functions, λ-terms, higher-order unification and polymorphic types. To achieve this, λprolog extends the classical first-order theory of Horn-clauses to the intuitionistic higher-order theory of *Hereditary Harrop* formulas [85].

The algebra of modular system (AMS) [119, 120] is a framework in which arbitrary logics with a model semantics can be combined. The difference with our work is that in AMS, connectives from the different logics cannot be combined arbitrarily. Instead, there is a fixed set of connectives (a "master" logic) that can be used to combine expressions from different logics. Compared to our logic, this has advantages and disadvantages. One advantage is that AMS only requires a two-valued semantics (an infon) to be specified for a given logic, making it more easily applicable to a wide range of logics. A disadvantage is that it does not allow for interactions between the different connectives.

3.2.2 Infons

The concept of infon in the sense used in this chapter is related to intensional objects in Montague's intensional logic [71]. Intensional logic studies the dichotomy between the designation (what they refer to) and the meaning of expressions. Intensional objects are represented by lambda expressions and model functions from states to objects similar to our infons. The term "infon" was used by other authors in other areas. In situation semantics [9], infons intuitively represent "quantums of information" [45]. Although such an infon has a different mathematical form than an infon in our theory, it determines a characteristic function from *situations* (which are approximate representations of states, similar to approximate structures) to true, false (or undetermined), which intuitively corresponds to an infon.

Situation semantics, the semantics supported by situation theory, provides a foundation for reasoning about real world situations and the derivations made by common sense. In [69], infons are "statements viewed as containers of information" and an (intuitionistic) logic of infons is built for the specific purpose of modelling distributed knowledge authorization.

3.3 Preliminaries

3.3.1 Cartesian product, powerset, product, pointwise extension and lifting

The *powerset* operator $\mathcal{P}(\cdot)$ maps a set X to its powerset $\mathcal{P}(X)$. The *power* operator $(\cdot)^{(\cdot)}$ maps pairs (I, Y) of sets to the set Y^I of all functions with domain I and co-domain Y. We denote the function with domain D and co-domain C that maps elements $x \in D$ to the value of a mathematical expression $exp[x]$ in variable x as $\lambda : D \to C : x \mapsto exp[x]$ (using λ as the anonymous function symbol as in lambda calculus). Or, if the co-domain is clear from the context, as $\lambda x \in D : exp[x]$. When $exp[x]$ is a Boolean expression, this is also denoted as a set comprehension $\{x \in D \mid exp[x]\}$.

We define the set of truth values $\mathcal{T}wo = \{\mathbf{f}, \mathbf{t}\}$; here $\mathbf{t}$ stands for "true" and $\mathbf{f}$ for "false". For any X, $\mathcal{P}(X)$ is isomorphic to $\mathcal{T}wo^X$, using the mapping from a set to its characteristic function. In the rest of the chapter, we will identify $\mathcal{P}(\cdot)$ with $\mathcal{T}wo^{(\cdot)}$.

We frequently use $\langle x_i \rangle_{i \in I}$ to denote the function $\lambda : I \to \{x_i \mid i \in I\} : i \to x_i$. We call this an indexed set (with index set I). Let $\langle V_i \rangle_{i \in I}$ be an indexed set of sets, i.e., each V_i is a set. Its *product set*, denoted $\times_{i \in I} V_i$, is the set of all indexed sets $\langle x_i \rangle_{i \in I}$ such that $x_i \in V_i$ for each $i \in I$. This generalizes Cartesian product $V_1 \times \cdots \times V_n$ (taking $I = \{1, \ldots, n\}$).

Let $\langle \leq_i \rangle_{i \in I}$ be an indexed set of partial order relations $\leq_i$ on sets V_i for each $i \in I$. The *product order* of $\langle \leq_i \rangle_{i \in I}$ is the binary relation

$$\{(\langle v_i \rangle_{i \in I}, \langle w_i \rangle_{i \in I}) \in (\times_{i \in I} V_i)^2 \mid \forall i \in I : v_i \leq_i w_i\}$$

It is a binary relation on $\times_{i \in I} V_i$. Written differently, it is the Boolean function:

$$\lambda : (\times_{i \in I} V_i)^2 \to \mathcal{T}wo : (\langle v_i \rangle_{i \in I}, \langle w_i \rangle_{i \in I}) \mapsto \wedge_{i \in I}(v_i \leq_i w_i)$$

A special case is if all V_i and $\leq_i$ are the same, i.e., for some V and $\leq$, it holds that $V_i = V$ and $\leq_i = \leq$ for each $i \in I$. Then the product relation $\times_{i \in I} \leq$ will be called the *pointwise extension* of $\leq$ on $V^I = \times_{i \in I} V$. Taking products of orders preserves many good properties of its component orders. It is well-known that the product order is a partial order. The product order of chain complete orders is a chain complete order and the product order of complete lattice orders is a complete lattice order.

Let $\langle O_i \rangle_{i \in I}$ be an indexed set of operators $O_i \in X_i^{V_i}$. Then we define the lift operator $\uparrow_{i \in I} O_i$ as the operator in $(\times_{i \in I} X_i)^{(\times_{i \in I} V_i)}$ that maps elements $\langle v_i \rangle_{i \in I}$

to $\langle O_i(v_i)\rangle_{i\in I}$. In another notation, it is the function:

$$\lambda : \times_{i\in I} V_i \to \times_{i\in I} X_i : \langle v_i\rangle_{i\in I} \mapsto \langle O_i(v_i)\rangle_{i\in I}$$

A special case arises when all O_i are the same operator $O : V \to V$. In this case, $\uparrow_{i\in I} O$ is a function in $\times_{i\in I} V = V^I$ mapping $\langle v_i\rangle_{i\in I}$ to $\langle O(v_i)\rangle_{i\in I}$. That is, it is the function $\lambda : V^I \to V^I : f \mapsto O \circ f$. We call this the *lifting* of $O : V \to V$ to the product V^I.

3.3.2 (Approximation) Fixpoint Theory

A binary relation $\leq$ on set V is a *partial order* if $\leq$ is reflexive, transitive and asymmetric. In that case, we call the mathematical structure $\langle V, \leq\rangle$ a *poset*. $\leq$ is *total* if for every $x, y \in V$, $x \leq y$ or $y \leq x$. The partial order $\leq$ is a *complete lattice order* if for each $X \subseteq V$, there exists a least upperbound $lub(X)$ and a greatest lower bound $glb_{\leq_p}(X)$. If $\leq$ is a complete lattice order of V, then V has a least element $\bot$ and a greatest element $\top$.

Let $\langle V, \leq\rangle, \langle W, \leq\rangle$ be two posets. An operator $O : V \to W$ is monotone if it is order preserving; i.e., if $x \leq y \in V$ implies $O(x) \leq O(y)$.

Let $\langle V, \leq\rangle$ be complete lattice with least element $\bot$ and greatest element $\top$. Its bilattice is the structure $\langle V^2, \leq_p, \leq\rangle$ with $(v_1, v_2) \leq_p (w_1, w_2)$ if $v_1 \leq w_1, v_2 \geq w_2$ and $(v_1, v_2) \leq (w_1, w_2)$ if $v_1 \leq w_1, v_2 \leq w_2$. The latter is the pointwise extension of $\leq$ to the bilattice. Both orders are known to be lattice orders. $\leq_p$ is called the *precision order*. The least precise element is $(\bot, \top)$ and most precise element is $(\top, \bot)$. An *exact* pair is of the form (v, v). A *consistent* pair (v, w) is one such that $v \leq w$. We say that (v, w) approximates $u \in V$ if $v \leq u \leq w$. The set of values approximated by (v, w) is $[v, w]$. This set is non-empty iff (v, w) is consistent. Exact pairs (V, V) are the maximally consistent pairs and they approximate a singleton $\{X\}$. We view the exact pairs as the embedding of V in V^2. Abusing this, we sometimes write v where (v, v) should be written. Pairs $(v, w) \in V^2$ are written as $\mathfrak{v}$, with $(\mathfrak{v})_1 = v, (\mathfrak{v})_2 = w$.

We define $V^c = \{(v, w) \in V^2 \mid v \leq w\}$. It is the set of consistent pairs. We will call such a pair an approximate value, and we call V^c the *approximate value space* of V.

It can be shown that any non-empty set $X \subseteq V^c$ has a greatest lower bound $glb(X)$ in V^c, but not every set $X \subseteq V^c$ has a least upperbound in V^c. In particular, the exact elements are exactly the maximally precise elements. Hence, V^c is not a complete lattice. However, if X has an upperbound in V^c, then $lub(X)$ exists. Also, V^c is chain complete: every totally ordered subset $X \subseteq V^c$

has a least upperbound. It follows that each sequence $\langle (v_i, w_i)\rangle_{i<\alpha}$ of increasing precision has a least upperbound $lub(\langle (v_i, w_i)\rangle_{i<\alpha})$, called its limit. This suffices to warrant the existence of a least fixpoint for every $\leq_p$-monotone operator $O : V^c \to V^c$.

Example 3.3.1. Consider the lattice $\mathcal{T}wo = \{\mathbf{t}, \mathbf{f}\}$ with $\mathbf{f} \leq \mathbf{t}$. The four pairs of its billatice $\mathcal{F}our$ correspond to the standard truth values of four-valued logic. The pairs $(\mathbf{t}, \mathbf{t})$ and $(\mathbf{f}, \mathbf{f})$ are the embeddings of true $(\mathbf{t})$ and false $(\mathbf{f})$ respectively. The pair $(\mathbf{f}, \mathbf{t})$ represents unknown $(\mathbf{u})$ and $(\mathbf{t}, \mathbf{f})$ represents the inconsistent value $(\mathbf{i})$. Here, the set $\mathcal{T}wo^c$ is the set of consistent pairs and is denoted $\mathcal{T}hree$. The precision order is $\mathbf{u} \leq_p \mathbf{t} \leq_p \mathbf{i}, \mathbf{u} \leq_p \mathbf{f} \leq_p \mathbf{i}$ and the product order is $\mathbf{f} \leq \mathbf{u} \leq \mathbf{t}, \mathbf{f} \leq \mathbf{i} \leq \mathbf{t}$. ▲

For any lattice $\langle V, \leq\rangle$ and domain D, the pointwise extension of $\leq$ to V^D is a lattice order, also denoted as $\leq$. The lattice V^D has a bilattice $(V^D)^2$ and approximate value space $(V^D)^c$ which are isomorphic to $(V^2)^D$, respectively $(V^c)^D$.

Example 3.3.2. The billattice of $\mathcal{T}wo^D$ and the approximation space $(\mathcal{T}wo^D)^c$ are isomorphic to $\mathcal{F}our^D$, respectively $\mathcal{T}hree^D$ under the pointwise extensions of $\leq_p$ and $\leq$ of $\mathcal{F}our$ and $\mathcal{T}hree$. Elements of $\mathcal{F}our^D$ and $\mathcal{T}hree^D$ correspond to four and three valued sets. ▲

Example 3.3.3. Take $D = \{a, b\}$. An element of $(\mathcal{T}wo^D)^2$ is the pair $(\{a \mapsto t, b \mapsto f\}, \{a \mapsto t, b \mapsto t\})$. This is an approximating element. The corresponding 3-valued set is $\{a \mapsto u, b \mapsto t\})$. ▲

Let D, C be complete lattices.

Definition 3.3.4. For any function $f : D \to C$, we say that $A : D^c \to C^c$ is an *approximator* of f if (1) ($\leq_p$-monotonicity) A is $\leq_p$-monotone and (2) (approximation) for each $v \in D$, $A(v) \leq_p f(v)$. We call A *exact* if A preserves exactness. The projections of $A(v, w)$ on first and second argument are denoted $A(v, w)_1$ and $A(v, w)_2$.

Approximators of f allow to infer approximate output from approximate input for f. The co-domain of an approximator is equipped with a precision order which can be pointwise extended on $(C^c)^{D^c}$.

Definition 3.3.5. We say that F is the *ultimate approximator* of f if F is the most precise approximator of f. We denote F as $\lceil f \rceil$.

The set of approximators for $D \to C$ is a subset of $(C^c)^{D^c}$. The co-domain C^c of an approximator is equipped with a precision order. Its product extension

to $(C^c)^{D^c}$ is chain-complete as well. It can be shown that for each $f : D \to C$, the set of approximators of f has a most precies element defined as $\lceil f \rceil(\mathfrak{v}) = glb_{\leq_p}(\{f(v) \mid \mathfrak{v} \leq_p v \in D\})$ [41].

Example 3.3.6. The ultimate approximators of the standard Boolean functions $\wedge, \neg, \vee, \ldots$, correspond to the standard 3-valued Boolean extensions known from the Kleene truth assignment. E.g., $\lceil \wedge \rceil$:

$\lceil \wedge \rceil$	**f**	**u**	**t**
f	**f**	**f**	**f**
u	**f**	**u**	**u**
t	**f**	**u**	**t**

▲

Let $\langle V, \leq \rangle$ be complete lattice with least element $\bot$ and greatest element $\top$. With an operator $O : V \to V$, many sorts of fixpoints can be associated: the standard fixpoints $O(x) = x$ and the *grounded* fixpoints of O [12]. For any approximator $A : V^c \to V^c$, more sorts of fixpoints can be defined:

- The A-Kripke-Kleene fixpoint is the $\leq_p$-least fixpoint of A.
- A partial A-stable fixpoint is a pair (x, y) such that
  - $A(x, y) = (x, y)$,
  - (x, y) is prudent, i.e., for all $z \leq y$, $A(z, y)_1 \leq y$ implies $x \leq z$.
  - there is no $z \in [x, y[$ such that $A(x, z)_2 \leq z$.
- The well-founded fixpoint of A is the least precise A-partial stable fixpoint.
- An A-stable fixpoint is an element $v \in L$ such that (v, v) is a partial A-stable fixpoint.

Assume A approximates O. It is well-known that the KK-fixpoint of A approximates all fixpoints of O and all partial stable fixpoints of A, hence also the well-founded fixpoint of A and the (exact) stable fixpoints of A. It can be shown that the three-valued immediate consequence operator of logic programs is an approximator of the two-valued one, and that the above sorts of fixpoints induce the different sorts of semantics of logic programming [40].

With a lattice operator $O : V \to V$, we define the ultimate well-founded fixpoint and the ultimate (partial) stable fixpoints as the well-founded fixpoint and the (partial) stable fixpoints of $\lceil O \rceil$. Compared with other approximators A of O, the ultimate approximator has the most precise KK-fixpoint and well-founded

fixpoint, and -somewhat surprisingly- the most (exact) stable fixpoints. That is, the set of exact stable fixpoints of any approximator A of O is a subset of that set of $\lceil O \rceil$. Notice that the ultimate well-founded fixpoint of O is an element of the bilattice, but it may be (and often is) exact.

In the context of logic programming a logic program Π characterises a 3-valued immediate consequence operator $\Phi : H_3 \to H_3$ where H_3 is the set of 3-valued Herbrand interpretations: $H_3 = Three^{HB}$ where HB is the Herbrand Base. It was shown in [41] that the Kripke Kleene model of Π is the Φ_Π Kripke Kleene fixpoint. That (partial) stable models of Π are Φ_Π stable fixpoints and that the well-founded model of Π is the Φ_Π well-founded fixpoint of Φ_Π.

3.4 A typed higher order logic $\mathbb{L}$ with (nested) definitions

3.4.1 Type system

A typed logic $\mathbb{L}$ contains a type system, offering a method to expand arbitrary sets $\mathbb{B}$ of (user-defined) type symbols to a set $\mathbb{T}(\mathbb{B})$ of types, together with a method to expand a type structure $\mathcal{A}$ assigning sets of values to the symbols of $\mathbb{B}$, to an assignment $\bar{\mathcal{A}}$ of sets of values to all types in $\mathbb{T}(\mathbb{B})$. We formalize these concepts.

Definition 3.4.1. A type vocabulary $\mathbb{B}$ is a (finite) set of type symbols. A type structure $\mathcal{A}$ for $\mathbb{B}$ is an assignment of sets $\tau^{\mathcal{A}}$ to each $\tau \in \mathbb{B}$.

Definition 3.4.2. A *type constructor* is a pair (tc, Sem_{tc}) of a type constructor symbol tc of some arity $n \geq 0$ and its associated semantic function Sem_{tc} which maps n-tuples of sets to sets such that Sem_{tc} preserves set isomorphism. [1]

Given a set $\mathbb{B}$ of type symbols and a set of type constructor symbols, a set of (finite) type terms τ can be built from them. In general, the set $\mathbb{T}(\mathbb{B})$ of types of a logic theory form a subset of the set of these type terms.

Definition 3.4.3. A type system consists of a set of type constructors and a function mapping any set $\mathbb{B}$ of type symbols to a set $\mathbb{T}(\mathbb{B})$ of type terms formed from $\mathbb{B}$ and the type constructor symbols such that for any bijective renaming $\theta : \mathbb{B} \to \mathbb{B}'$, $\mathbb{T}(\mathbb{B})$ and $\mathbb{T}(\mathbb{B}')$ are identical modulo the renaming θ. An element of $\mathbb{T}(\mathbb{B})$ is called a *type*. A *compound type* is an element of $\mathbb{T}(\mathbb{B}) \setminus \mathbb{B}$.

[1] That is, if there exists bijections between S_1 and S'_1, ..., S_n and S'_n, then there is a bijection between $Sem_{tc}(S_1, \ldots, S_n)$ and $Sem_{tc}(S'_1, \ldots, S'_n)$.

For a given type system, it is clear that any type structure $\mathcal{A}$ for $\mathbb{B}$ can be expanded in a unique way to all type terms by iterated application of the semantic functions Sem_{tc}.

Definition 3.4.4. Given a type system and a type structure $\mathcal{A}$ for a set $\mathbb{B}$ of type symbols, we define $\bar{\mathcal{A}}$ as the unique expansion of $\mathcal{A}$ to $\mathbb{T}(\mathbb{B})$ defined by induction on the structure of type terms and using the semantic type constructor functions Sem_{tc} of type constructors.

By slight abuse of notation, we write $\tau^{\bar{\mathcal{A}}}$ as $\tau^{\mathcal{A}}$.

Definition 3.4.5. We call a type system *type closed* if for every $\mathbb{B}$, $\mathbb{T}(\mathbb{B})$ is the set of all type terms built over $\mathbb{B}$ and the type constructors of the system.

Example 3.4.6. The type system of the logic that we will define below is type closed. Its type constructor symbols and corresponding semantic type operators are:

- the 0-ary Boolean type constructor symbol BOOL with $Sem_{\mathsf{BOOL}} = \mathcal{T}wo$;
- the 0-ary natural number constructor type symbol NAT with $Sem_{NAT} = \mathbb{N}$;
- the n-ary Cartesian product type constructor symbol $\times_n$; we write $\times_n(\tau_1, \ldots, \tau_n)$ as $\tau_1 \times \cdots \times \tau_n$ and $\times_n(\tau, \ldots, \tau)$ as τ^n. The semantic operator $Sem_{\times_n}$ maps tuples of sets $(S_1, \ldots, S_n)$ to the Cartesian product $S_1 \times \cdots \times S_n$;
- the function type constructor $\to$ with $Sem_{\to}$ mapping pairs of sets (X, Y) to the function set Y^X. Note that we only consider total functions.

In typed lambda calculus, Cartesian product is often not used (it can be simulated using higher order functions and *currying*). Here, we keep it in the language to connect easier with FO. ▲

Example 3.4.7. The type system of typed classical first order logic uses the type constructors corresponding to BOOL, $\times_n$ and $\to$ in the previous example. $\mathbb{T}(\mathbb{B})$ is the set $\{\tau_1 \times \cdots \times \tau_n \to \mathsf{BOOL}, \tau_1 \times \cdots \times \tau_n \to \tau \mid \tau_1, \ldots, \tau_n, \tau \in \mathbb{B}\}$. It consists of first order predicate types $\tau_1 \times \cdots \times \tau_n \to \mathsf{BOOL}$ and first order function types $\tau_1 \times \cdots \times \tau_n \to \tau$. The type system of untyped classical first order logic is obtained by fixing $\mathbb{B} = \{U\}$, where U represents the universe of discourse. Clearly, (typed) FO is not type closed. ▲

From now on, we assume a fixed type system. We also assume an infinite supply of type symbols, and for all types τ that can be constructed from this supply

and the given type constructor symbols, an infinite supply of symbols σ of type τ. We write $\sigma : \tau$ to denote that τ is the type of σ.

Definition 3.4.8. A *vocabulary* (or *signature*) Σ is a tuple $\langle \mathbb{B}, Sym \rangle$ with $\mathbb{B}$ a set of type symbols, Sym a set of symbols σ of type $\tau \in \mathbb{T}(\mathbb{B})$.

We write $\mathbb{T}(\Sigma)$ to denote $\mathbb{T}(\mathbb{B})$.

Let Σ be a vocabulary $\langle \mathbb{B}, Sym \rangle$.

Definition 3.4.9. An *assignment* to Sym in type structure $\mathcal{A}$ for $\mathbb{B}$ is a mapping $A : Sym \to \bigcup\{\tau^{\mathcal{A}} \mid \tau \in \mathbb{T}(\Sigma)\}$ such that for each $\sigma : \tau \in Sym$, $\sigma^A \in \tau^{\mathcal{A}}$. That is, the value of σ^A is of type τ in $\mathcal{A}$. The set of assignments to Sym in $\mathcal{A}$ is denoted $Assign^{\mathcal{A}}_{Sym}$.

Definition 3.4.10. A Σ-*structure* I is a tuple $\langle \mathcal{A}, (\cdot)^I \rangle$ of a type structure $\mathcal{A}$ for $\mathbb{B}$, and $(\cdot)^I$ an *assignment* to all symbols $\sigma \in Sym$ in type structure $\mathcal{A}$. We denote the value of σ as σ^I. The class of all Σ-structures is denoted $\mathbb{S}(\Sigma)$.

We frequently replace $\mathcal{A}$ by I; e.g., we may write τ^I for $\tau^{\mathcal{A}}$.

Let Σ be a vocabulary with type symbols $\mathbb{B}$, I a Σ-structure. Let Sym be a set of symbols with types in $\mathbb{T}(\mathbb{B})$ (it may contain symbols not in Σ). For any assignment $A \in Assign^I_{Sym}$ to Sym in (the type structure of) I, we denote by $I[A]$ the structure that is identical to I except that for every $\sigma \in Sym$, $\sigma^{I[A]} = \sigma^A$. This is a structure of the vocabulary $\Sigma \cup Sym$. As a shorthand notation, let σ be a symbol of type τ and v a value of type τ in I, then $[\sigma : v]$ is the assignment that maps σ to v, and $I[\sigma : v]$ is the updated structure.

Definition 3.4.11. A Σ-*infon* $\mathfrak{i}$ of type $\tau \in \mathbb{T}(\Sigma)$ is a mapping that associates with each Σ-structure I a value $\mathfrak{i}(I)$ of type τ in I. The class of Σ-infons is denoted $\mathfrak{I}_\Sigma$. Each symbol $\sigma \in \Sigma$ of type τ defines the Σ-infon $\mathfrak{i}_\sigma$ of type τ that associates with each Σ-structure I the value σ^I.

Infons of type τ are similar to intensional objects in Montague's intensional logic [71]. An infon of type BOOL provides an abstract syntax independent representation of a *quantum of information*. Structures representing a possible state of affairs in which the information holds is mapped to true by the infon, while structures in which the information does not hold are mapped to false. It will be the case that two sentences are logically equivalent in the standard sense iff they induce the same infon.

3.4.2 Language constructs

Definition 3.4.12. A language construct $\mathfrak{C}$ consists of an arity n representing the number of arguments, a typing rule $Type_{\mathfrak{C}}$ specifying the allowable argument types and the corresponding expression type, and a semantic operator $Sem_{\mathfrak{C}}$. A *typing rule* $Type_{\mathfrak{C}}$ is a partial function from n argument types $\tau_1, \ldots, \tau_n$ to a type $Type_{\mathfrak{C}}(\tau_1, \ldots, \tau_n) = \tau$ that preserves renaming of type symbols; i.e., if θ is a bijective renaming of type symbols, then $Type_{\mathfrak{C}}(\theta(\tau_1), \ldots, \theta(\tau_n)) = \theta(\tau)$. If $Type_{\mathfrak{C}}$ is defined for $\tau_1, \ldots, \tau_n$, we call $\tau_1, \ldots, \tau_n$ an argument type for $\mathfrak{C}$. The semantic operator $Sem_{\mathfrak{C}}$ is a partial mapping defined for all tuples of infons $\mathfrak{i}_1, \ldots, \mathfrak{i}_n$ of all argument types $\tau_1, \ldots, \tau_n$ for $\mathfrak{C}$ to an infon of the corresponding expression type τ.

A language construct $\mathfrak{C}$ takes a sequence of expressions $e_1, \ldots, e_n$ as argument and yields the compound expression $\mathfrak{C}(e_1, \ldots, e_n)$. This determines the abstract syntax of expressions. We often specify a concrete syntax for $\mathfrak{C}$ (which often disagrees with the abstract syntax).

Let $\tau_1, \ldots, \tau_n$ be an argument type for $\mathfrak{C}$ yielding the expression type τ. Then for well-typed expressions $e_1, \ldots, e_n$ of respectively types $\tau_1, \ldots, \tau_n$, the (abstract) compound expression $\mathfrak{C}(e_1, \ldots, e_n)$ is well-typed and of type τ. Some language constructs are polymorphic and apply to expressions of many types. Others have a unique type for each argument.

Example 3.4.13. The tupling operator *TUP* is a polymorphic language construct that maps expressions $e_1, \ldots, e_n$ of arbitrary types $\tau_1, \ldots, \tau_n$ to the compound expression $TUP(e_1, \ldots, e_n)$ of type $\tau_1 \times \cdots \times \tau_n$. The concrete syntax is $(e_1, \ldots, e_n)$. $Type_{TUP}$ is defined on all type expressions.

The conjunction $\wedge$ maps expressions e_1, e_2 of type BOOL to $\wedge(e_1, e_2)$ of type BOOL. The concrete syntax is $e_1 \wedge e_2$. $Type_{\wedge}$ is only defined on all pairs of boolean expressions. ▲

The set of language constructs of a logic $\mathbb{L}$ together with a vocabulary Σ uniquely determines the set $Exp^{\mathbb{L}}_{\Sigma}$ of well-typed expressions over Σ, as well as a function $Type_{\mathbb{L}} : Exp^{\mathbb{L}}_{\Sigma} \to \mathbb{T}(\Sigma)$. Formally, consider the set of (finite) labeled trees with nodes labeled by language constructs of $\mathbb{L}$ and symbols of Σ. Within this set, the function $Type_{\mathbb{L}}$ is defined by induction on the structure of expressions

- $Type_{\mathbb{L}}(\sigma) = \tau$ if $\sigma \in \Sigma$ is a symbol of type τ;
- $Type_{\mathbb{L}}(\mathfrak{C}(e_1, \ldots, e_n)) = Type_{\mathfrak{C}}(Type_{\mathbb{L}}(e_1), \ldots, Type_{\mathbb{L}}(e_n))$.

This mapping $Type_{\mathbb{L}}$ is a partial function, the domain of which is exactly $Exp^{\mathbb{L}}_{\Sigma}$.

Furthermore, the set of language constructs of $\mathbb{L}$ determines for each well-typed expression $e \in Exp^{\mathbb{L}}_{\Sigma}$ of type τ a unique infon $Sem_{\mathbb{L}}(e)$ of that type. The function $Sem_{\mathbb{L}}$ is defined by induction on the structure of expressions by the following equation:

- $Sem_{\mathbb{L}}(\sigma) = \mathfrak{i}_{\sigma}$ if $\sigma \in \Sigma$;
- $Sem_{\mathbb{L}}(\mathfrak{C}(e_1, \ldots, e_n)) = Sem_{\mathfrak{C}}(Sem_{\mathbb{L}}(e_1), \ldots, Sem_{\mathbb{L}}(e_n))$.

This property warrants a strong form of Frege's compositionality principle.

We call a logic *substitution closed* if every expression of some type may occur at any argument position of that type. E.g., propositional logic and first order logic are substitution closed, but CNF is not due to the syntactical restrictions on the format of CNF formulas.

Simply typed lambda calculus with infon semantics

Below, we introduce a concrete substitution closed logic $\mathbb{L}$ with a type closed type system. We specify the main language constructs.

- $TUP(e_1, \ldots, e_n)$:
  - concrete syntax is $(e_1, \ldots, e_n)$;
  - typing rule: for arguments of types $\tau_1, \ldots, \tau_n$ respectively, the compound expression is of type $\tau_1 \times \cdots \times \tau_n$;
  - Sem_{TUP} maps finite tuples $\bar{\mathfrak{i}}$ to the infon: $\lambda I \in \mathbb{S}(\Sigma) : (\mathfrak{i}_1(I), \ldots, \mathfrak{i}_n(I))$.
- $APP(e, e_1)$:
  - concrete syntax $e(e_1)$;
  - typing rule: for arguments of type $\tau_1 \to \tau, \tau_1$, the expression is of type τ;
  - Sem_{APP}: maps well-typed infons $\mathfrak{i}, \mathfrak{i}_1$ to $\lambda I \in \mathbb{S}(\Sigma) : \mathfrak{i}(I)(\mathfrak{i}_1(I))$.
- $Lambda(\bar{\sigma}, e)$: here $\bar{\sigma}$ is a finite sequence $\sigma_1, \ldots, \sigma_n$ of symbols (not expressions);
  - concrete syntax $\lambda\sigma_1 \ldots \sigma_n : e$; if e is Boolean, then $\{\sigma_1 \ldots \sigma_n : e\}$;
  - typing rule: if the symbols $\sigma_1, \ldots, \sigma_n$ are of types $\tau_1, \ldots, \tau_n$ and the second argument is of type τ, the expression is of type $(\tau_1 \times \cdots \times \tau_n) \to \tau$;

– Sem_{Lambda} maps an $\Sigma \cup \{\sigma_1, \ldots, \sigma_n\}$-infon $\mathfrak{i}$ of type τ to the Σ-infon $\lambda I \in \mathbb{S}(\Sigma) : F_I$, where F_I is the function $\lambda \bar{x} \in {\tau_1}^I \times \cdots \times {\tau_n}^I : \mathfrak{i}(I[\bar{\sigma} : \bar{x}])$.

Equality, connectives and quantifiers are introduced using interpreted symbols, symbols with a fixed interpretation in each structure.

The logical symbols $\wedge, \vee : \mathsf{BOOL} \times \mathsf{BOOL} \to \mathsf{BOOL}$ and $\neg : \mathsf{BOOL} \to \mathsf{BOOL}$ have the standard Boolean functions as interpretations in every structure.

Quantifiers and equality are polymorphic. We introduce instantiations of them for all types τ. For every type τ, $\forall_\tau, \exists_\tau$ are symbols of type $(\tau \to \mathsf{BOOL}) \to \mathsf{BOOL}$. For concrete syntax, for $\forall_\tau(Lambda(\sigma, e))$ with e a Boolean expression and $\sigma : \tau$, we write $\forall \sigma : e$ (we dropped the underscore from $\forall_\tau$ since τ is the type of σ). It also corresponds to a quantified set comprehension $\forall_\tau(\{\sigma : \varphi\})$. In any structure I, ${\forall_\tau}^I$ is the Boolean function $\lambda X \in (\tau \to \mathsf{BOOL})^I : (X = \tau^I)$ that maps a set X with elements of type τ to $\mathbf{t}$ if X contains all elements of this type in I. Likewise, ${\exists_\tau}^I$ is the Boolean function $\lambda X \in (\tau \to \mathsf{BOOL})^I : (X \neq \emptyset)$.

Equality is a polymorphic interpreted predicate. For each τ, introduce a symbol $=_\tau$ of type $\tau \times \tau \to \mathsf{BOOL}$. The concrete syntax is $e = e_1$. Its interpretation in an arbitrary structure I is the identity relation of type τ^I.

Likewise, standard aggregate functions such as cardinality and sum are introduced as interpreted higher order Boolean functions. We introduce the interpreted symbol

$$Card_\tau : ((\tau \to \mathsf{BOOL}) \times NAT) \to \mathsf{BOOL}$$

interpreted in each structure I as the function

$${Card_\tau}^I : ((\tau \to \mathsf{BOOL})^I \times \mathbb{N}) \to \mathcal{T}wo : (S, n) \mapsto (\#(S) = n)).$$

We have chosen here to define $Card_\tau$ as a binary predicate symbol rather than as a unary function, because it is a partial function defined only on finite sets and our logic is not equipped for partial functions. A third approach would have been to extend the domain of natural numbers with infinity.

3.4.3 The definition construct DEF for higher order and nested definitions

So far, we have defined typed lambda calculus under an infon semantics. In this section, we extend the language with higher order versions of definitions as

in the logic FO(ID). There, definitions are conventionally written as finite set of rules $\forall\bar{\sigma}(P(\bar{\sigma}) \leftarrow \varphi)$ where $P : (\bar{\tau} \to \mathsf{BOOL})$ is a predicate symbol, $\bar{\sigma} : \bar{\tau}$ a (sequence of) symbol(s), and φ a Boolean expression. An example can be seen in Listing 3.1.

Listing 3.1: The transitive closure of G

```
{
∀x ∀y: Reach(x,y)← G(x,y).
∀x ∀z: Reach(x,z)← G(x,y)∧ Reach(y,z).
}
```

In the abstract syntax, a rule $\forall\bar{\sigma}(P(\bar{\sigma}) \leftarrow \varphi)$ will be represented as a pair $(P, \{\bar{\sigma} : \varphi\})$.

Below, we use the mathematical variable Δ to denote definition expressions. In general, an abstract expression of the definition construct DEF is of the form $DEF(\bar{P}, \bar{e})$ where $\bar{P}$ is a finite sequence $(P_1, \ldots, P_n)$ $(n > 0)$ of predicate symbols and $\bar{e}$ an equally long sequence of expressions. We write $(P, e) \in \Delta$ to denote that for some $i \leq n$, $P_i = P$ and $e_i = e$. Let $DP(\Delta)$ be $\{P_1, \ldots, P_n\}$, the set of *defined symbols* of Δ. It is possible that the same symbol P has multiple rules in Δ (as in the above example).

- For the concrete syntax, $DEF(\bar{P}, \bar{e})$ represents a definition with n rules corresponding to the pairs (P_i, e_i). If e_i is the set comprehension $\{\bar{\sigma} : \varphi\}$, the corresponding rule in concrete syntax is $\forall\bar{\sigma}(P_i(\bar{\sigma}) \leftarrow \varphi)$.

  Due to the substitution closedness of the logic, new abstract rules are allowed. E.g., $(Reach, G)$ is an abstract representation that is equivalent to the first rule in the *Reach* example, and it is an alternative way to represent the base case of the reachability relation.

- Typing rule: if for each $i \in [1, n]$, P_i, e_i are of the same type $\tau_i \to \mathsf{BOOL}$ then the definition expression is of type BOOL. It follows that the value of a definition in a structure is true or false. Note that defined symbols are predicate symbols.

- Sem_{DEF}: this operator maps tuples $((P_1, \ldots, P_n), (\mathfrak{i}_1, \ldots, \mathfrak{i}_n))$ where each $\mathfrak{i}_i$ is an infon of type τ_i to an infon $\mathfrak{i}$ of type BOOL. This operator will be applied to the infons $\mathfrak{i}_i$ of the expressions e_i. To define the infon $\mathfrak{i}$ from the input, we construct for each $I \in \mathbb{S}(\Sigma)$ the immediate consequence operator Γ^I_Δ.

  The operator Γ^I_Δ is an operator on $Assign^I_{DP(\Delta)}$, the lattice of $DP(\Delta)$-assignments in I. Note that for a rule $(P, e) \in D$, the value e^I of e in a

structure I is exactly the set that this rule produces for P in I. The total produced value for P is then obtained by taking the union of all rules defining P. Formally, for each $P \in DP(\Delta)$, let $INF_P = \{\mathfrak{i}_i \mid P_i = P\}$. That is, INF_P is the set of infons amongst $\mathfrak{i}_1, \ldots, \mathfrak{i}_n$ that correspond to rules with P in the head. Then Γ^I_Δ maps an assignment $A \in Assign^I_{DP}$ to an assignment B such that for each $P \in DP$:

$$P^B = lub_\leq(\{\mathfrak{i}_i(I[A]) \mid \mathfrak{i}_i \in INF_P\}$$

That is, P^B is the union of what each rule of P produces in the structure $I[A]$.

The operator Γ^I_Δ is well-defined, and indeed, it is the immediate consequence operator of Δ in structure I. This is a lattice operator on the lattice of assignments of the defined symbols $DP(\Delta)$ in I. Consequently, this operator will have an ultimate well-founded fixpoint UWF^I_Δ, the well-founded fixpoint of the ultimate approximator $\lceil \Gamma^I_\Delta \rceil$. This fixpoint may be exact or not. We define the truth value Δ^I of Δ in I as $(I = UWF^I_\Delta)$, that is, $\Delta^I = \mathbf{t}$ if I is the exact ultimate well-founded fixpoint of the operator, and $\Delta^I = \mathbf{f}$ otherwise. The infon $Sem_{\mathbb{L}}(\Delta)$ is the Boolean infon $\lambda I \in \mathbb{S}(\Sigma) : (I = UWF^I_\Delta)$.

The semantic operator $Sem_{\mathbb{L}}$ associates with each expression an infon, and with each theory T a Boolean infon $\mathfrak{i}$. This induces a model semantics, in particular $M \models T$ if $\mathfrak{i}(M) = \mathbf{t}$.

Theorem 3.4.14. *The logic FO(ID) equipped with the ultimate well-founded semantics for definitions is a fragment of $\mathbb{L}$. That is, any theory T of FO(ID) corresponds syntactically to one T' of $\mathbb{L}$ and T and T' have the same models (taking the ultimate well-founded semantics for definitions).*

Proof. (sketch) Note that for all logical connectives, the semantical rule directly corresponds to the rule for that connective in the truth evaluation function of FO(ID). In FO(ID) quantification binds a new identifier, but in $\mathbb{L}$ this corresponds to the application of the quantifier function to a lambda expressions which binds this identifier. Through this correspondence, the semantics of quantifiers are defined in the same way. From this we can conclude that FO is properly captured within $\mathbb{L}$.

This leaves us with the addition of inductive definitions. The semantics of inductive definitions for both FO(ID) and $\mathbb{L}$ are captured in Approximation Fixpoint Theory. These semantics only depend on the semantics of the bodies of the rules in the definition. The bodies of the rules of inductive definitions in FO(ID) are FO-formulas. We already established that these formulas were

properly captured. So from this we can derive that also the inductive definitions in FO(ID) and $\mathbb{L}$ have the semantics. □

3.4.4 Applications for Higher Order Definitions

Higher order definitions are natural representations for some complex concepts. A standard example is a definition of winning positions in two-player games as can be seen in Listing 3.2. This definition of `win` and `lose` is a monotone second order definition that uses simultaneous definition and has a two-valued well-founded model.

Listing 3.2: `cur` is a winning position in a two-player game

```
{
∀cur ∀Move ∀IsWon: win(cur, Move, IsWon) ← IsWon(cur) ∨
  ∃ nxt : Move(cur,nxt) ∧ lose(nxt,Move,IsWon).
∀cur ∀Move ∀IsWon: lose(cur,Move, IsWon) ← ¬IsWon(cur) ∧
  ∀ nxt : Move(cur,nxt) ⇒ win(nxt,Move,IsWon).
}
```

3.4.5 Templates

In Chapter 2, a subclass of higher order definitions were defined as templates. These templates allow us to define an abstract concept in an isolation, so that it can be reused multiple times. This prevents code duplication and results in more readable specifications. In the same context, we identified applications for nested definitions. An example of this can be seen in Listing 3.3. In that example a binary higher order predicate `tc` is defined, such that `tc(P,Q)` holds iff `Q` is defined as the transitive closure of `P`.

Listing 3.3: Q is the transitive closure of P

```
{
∀Q ∀P: tc(P,Q) ←
  {∀x ∀y: Q(x,y) ← P(x,y) ∨(∃ z: Q(x,z)∧Q(z,y))}.
}
```

Note that using this definition of `tc`, the definition in Listing 3.1 can simply be replaced with the atom `tc(Reach,G)`. This demonstrates the abstraction power of these definitions.

3.4.6 Graph Morphisms

A labeled graph is a tuple of a set of vertices, a set of edges between these vertices, and a labeling function on these vertices. Many applications work with labeled graphs: one example is the graph mining problem [68], which requires the notion of homomorphisms and isomorphisms between graphs. As other applications require these same concepts, these concepts lend themselves to a definition in isolation.

To achieve this, we first define the `graph` type as an alias for the higher order type $\mathcal{P}(node) \times \mathcal{P}(node \times node) \times \mathcal{P}(node \to label)$, where the components of the triple are called `Vertex`, `Edge` and `Label` respectively. To define when two graphs are `homomorph` and `isomorph`, we first define a helper predicate `homomorphism`. This predicate takes a function and two graphs, and is true when this function represents a homomorphism from the first graph to the second. We then define `homomorph` and `isomorph` in terms of the `homomorphism` predicate. In Listing 3.4, these higher order predicates are defined using higher order definitions. The higher order arguments of these definitions are either decomposed into the different tuple elements using matching (Line 3) or accepted as a single entity (Line 8).

Listing 3.4: Defining `homomorph` and `isomorph`

```
1  {
2  ∀F ∀V1 ∀Edge1 ∀Label1 ∀V2 ∀Edge2 ∀Label2:
3  homomorphism(F, (V1, Edge1, Label1), (V2, Edge2, Label2)) ←
4       (∀ x, y : Edge1(x, y) ⇒ Edge2(F(x), F(y))) ∧
5       (∀ x : Label1(x) = Label2(F(x)).
6
7  ∀G1 ∀G2:
8  homomorph(G1,G2) ←
9      ∃ F : homomorphism(F, G1, G2).
10
11 ∀G1 ∀G2:
12 isomorph(G1, G2) ←
13     (∃ F, G : (∀ x : G(F(x)) = x) ∧
14       homomorphism(F, G1, G2) ∧ homomorphism(G, G2, G1) ).
15 }
```

3.5 Conclusion

We defined a logic integrating typed higher order lambda calculus with definitions. The logic is type closed and substitution closed, allows definitions of higher order predicates and nested definitions. The logic satisfies a strong form of Frege's compositionality principle. The used principles also allow the definition of rules under other semantics (e.g., stable semantics). For future work, one question is how to define standard well-founded semantics for definitions in $\mathbb{L}$ rather than the ultimate well-founded semantics. It is well-known that both semantics often coincide, e.g., when the standard well-founded model is two-valued, which is frequently the case when rule sets are intended to express definitions of concepts. Nevertheless, standard well-founded semantics is computationally cheaper and is easier to implement. This provides a good motivation. Another question is how to extend definitions for arbitrary symbols, that is, for functions.

Part II

Implementing a Lambda Calculus in Answer Set Programming

About This Part

In the first part new logics were developed with much consideration for their semantics. More practical aspects, such as how a system using such a logic could be developed were mostly disregarded.

The most common practical uses for logics such as the ones developed in Part I are model expansion problems. Answer Set Programming systems and IDP3 are prime examples of this. Both ASP and IDP3 reduce the model expansion problem to a problem of finding a satisfying assignment for a SAT formula. To achieve this, they do a step called *grounding*, in which the full logic is reduced to a simpler propositional one.

There has been a lot of research investigating how grounding can be done efficiently. Therefore it can be beneficial, when implementing new logics, to reuse existing grounders and transform more complex logics to preexisting logics instead of directly to a propositional format such as SAT.

This second part of the thesis will take exactly this approach. It consists of two chapters. Chapter 4 shows how Programming Computable Functions (a simple lambda calculus) can be translated to Answer Set Programming. The techniques in this chapter can then be applied to translate lambda calculi with more language constructs to Answer Set Programming to obtain a system which allows a user to do model expansion on logics such as the one introduced in Chapter 3.

Chapter 5 then continues on this approach by explaining a prototype implementation of such a translator for a complex logic.

4 Translating Programming Computable Functions to Answer Set Programming

A first version of the work of this chapter were presented at the International Workshop on Functional and Logic Programming 2018 [30].

4.1 Introduction

A lot of research aims to put more abstraction layers into modelling languages for model expansion problems. One common approach is to add templates or macros to a language to enable the reuse of a concept [34, 35, 18]. Some languages such as HiLog [21] introduce higher order language constructs with first order semantics to mimic this kind of features. While the lambda calculus is generally not considered a regular modelling language, one of its strengths is the ability to easily define abstractions. We aim to combine both strengths by showing how the lambda calculus can be translated into existing paradigms. Our end goal is to leverage existing search technology for logic programming languages to serve as a search engine for problems specified in functional languages.

In this chapter we introduce a transpiling algorithm between Programming Computable Functions, a programming language based on the lambda calculus and Answer Set Programming, a logic-based modelling language. This transpilation is a source-to-source translation of programs. We show how this can be the basis for a functional modelling language which combines the advantages of easy abstractions and reasoning over non-defined symbols. Transpiling Programmable Computable Functions to other languages like C [66], or a theorem prover like Coq [29] has been done before. But as far as the authors are aware, no approaches to translate it into logic programs have been

developed.

In Section 4.2 we introduce the source language of our transpiler: Programming Computable Functions. In Section 4.3 we introduce the target language of our transpiler: Answer Set Programming. In Section 4.4 we describe the translation algorithm. Finally, in Section 4.5 we motivate why this kind of translation can be useful in practice. An implementation of the translator is made available online (`https://dtai.cs.kuleuven.be/krr/pcf2asp`).

4.2 Programming Computable Functions

Programming Computable Functions [47, 107] (PCF) is a programming language based on the lambda calculus. It is not used as an end-user language; instead it provides a strong theoretical basis for more elaborate languages, such as Lisp, Caml or Haskell. There are many small variations of PCF, such as Mini-ML[22]. Some of these variations extend it with booleans, tuples or arithmetic operators. The particular flavor is irrelevant for the principles in this chapter.

4.2.1 Syntax

The syntax of PCF relies heavily on the standard lambda calculus, extended with natural numbers, a selection construct and a fixpoint operator. We identify the following language constructs:

- function application $e_1 e_2$, which is left associative, this means that $e_1 e_2 e_3 = (e_1 e_2) e_3$,
- a lambda abstraction $\lambda \texttt{x}.e$, abstracting the variable `x` out of the expression `e`,
- for each numeral $n \in \mathbb{N}$, a constant `n`,
- constants `succ`, representing the successor function over $\mathbb{N}$, `pred` representing the predecessor function over $\mathbb{N}$,
- a constant `fix`, representing the fixpoint operator, also known as the Y-combinator, and
- a ternary language construct `ifz` e_z `then` e_t `else` e_e, representing an if zero-then-else.

Suppose that $\mathbb{I}$ is an infinite supply of identifiers. The syntax of PCF can be inductively defined as:

```
e = x (∈ 𝕀)  | e e  | λx.e
  | n (∈ ℕ) | succ | pred | fix | ifz e then e else e
```

Example 4.2.1. `(λx. succ (succ x)) (succ 0)` is a complicated way to write the number 3. ▲

The expression `fix` allows us to write functions which would require recursive definitions in most programming languages. It takes a function f as argument and returns the fixpoint x of that function so that $f(x) = x$. From this it follows that `fix` satisfies the equation `fix` f `=` f `(fix` f`)`.

Example 4.2.2. A traditional recursive definition for the double of a number `x` could be:

```
double x = ifz x then 0 else succ (succ (double (pred x)))
```

It is possible to rewrite this using fix, by abstracting both `double` and `x`, and using `pred` and `succ` for the increments and decrements:

```
fix (λdouble. λx. ifz x then 0 else succ (succ (double (pred x))
```

The informal meaning of this expression is the doubling function. ▲

Example 4.2.3. `fix (λplus. λa. λb. ifz a then b else succ (plus (pred a) b))` is a PCF term that can be informally read as the binary sum over integers. ▲

4.2.2 Operational Semantics

When considering expressions, we traditionally consider only those without free variables. However, when considering the operational semantics, we will generalise this to situations where free variables can occur. For this reason we introduce environments and closures through a mutually inductive definition.

Definition 4.2.4. An *environment* E is a mapping from identifiers to closures. A *closure* (E,e) consists of an environment E and an expression e, where the environment must interpret at least all the free variables in e.

We say an environment interprets an identifier x if it contains a mapping for x. The closure to which E maps an interpreted variable x is written as E[x].

Example 4.2.5. ({`a` $\mapsto$({},`succ 0`)},`succ a`) is a valid closure which will evaluate to the number 2. ▲

Evaluation context

The evaluation relation $\Downarrow$ is a relation between closures and values, which we will write as follows:

$$\mathrm{E},\ e\ \Downarrow\ \mathrm{V}.$$

(E,e) is the closure that is being evaluated. When considering the evaluation of an expression without an explicit environment, we assume it has no free variables and we interpret this is as the closure with the empty environment.

V is the value that corresponds to the expression, this can either be a natural number or a closure. A natural number can be implicitly used as a closure with the empty environment.

The evaluation relation will be defined using a higher order monotone inductive definition in the space of closures, expressions and values.

Notation

We will describe both the semantics of PCF and the translation algorithm using a set of inference rules. These are rules of the form

$$\frac{\mathrm{Premise}_1 \qquad \ldots \qquad \mathrm{Premise}_n}{\mathrm{Conclusion}}$$

An algorithmic interpretation of these rules will lead to a program which can evaluate/translate PCF. Most often, the easiest way to read this kind of rules is bottom up.

Evaluation Rules

The following inference rules determine the operational semantics for PCF through the evaluation relation $\Downarrow$.

$$\frac{}{\mathrm{E},\ \mathsf{x}\ \Downarrow\ \mathrm{E}[\mathsf{x}]}\ (\text{lookup})$$

$$\frac{\mathrm{E},\ e_1\ \Downarrow (\mathrm{E}_2,\ \lambda \mathtt{x}.e_3) \qquad \mathrm{E},\ e_2\ \Downarrow V \qquad \mathrm{E}_2 \cup \{\mathtt{x} \mapsto V\},\ e_3\ \Downarrow V_{ap}}{\mathrm{E},\ e_1 e_2\ \Downarrow V_{ap}}\ (\text{app})$$

$$\frac{}{\mathrm{E},\ \lambda \mathtt{x}.f\ \Downarrow (\mathrm{E},\ \lambda \mathtt{x}.f)}\ (\text{closure})$$

$$\frac{}{\mathrm{E},\ n(\in \mathbb{N})\ \Downarrow\ n}\ (\text{nat})$$

$$\frac{\mathrm{E},\ e\ \Downarrow n}{E,\ \mathtt{succ}\ e\ \Downarrow n+1}\ (\text{succ})$$

$$\frac{\mathrm{E},\ e\ \Downarrow n+1}{\mathrm{E},\ \mathtt{pred}\ e\ \Downarrow n}\ (\text{pred})$$

$$\frac{\mathrm{E},\ e_z \Downarrow 0 \qquad \mathrm{E},\ e_t \Downarrow V}{\mathrm{E},\ \mathtt{ifz}\ e_z\ \ \mathtt{then}\ e_t\ \ \mathtt{else}\ e_e\ \Downarrow\ V}\ (\text{if-then})$$

$$\frac{\mathrm{E},\ e_z \Downarrow n \qquad n > 0 \qquad \mathrm{E},\ e_e \Downarrow V}{\mathrm{E},\ \mathtt{ifz}\ e_z\ \mathtt{then}\ e_t\ \mathtt{else}\ e_e\ \Downarrow\ V}\ (\text{if-else})$$

$$\frac{\mathrm{E} \cup \{\mathtt{x} \mapsto (\mathrm{E},\ \mathtt{fix}\ (\lambda \mathtt{x}.e))\},\ e\ \Downarrow V}{\mathrm{E},\ \mathtt{fix}\ (\lambda \mathtt{x}.e)\ \Downarrow V}\ (\text{fix})$$

These rules form an inductive definition of the evaluation relation $\Downarrow$. Note that this is a call-by-value semantics. This can be seen in the rule of applications, as the subexpression e_2 is evaluated before adding it to the environment. A call-by-name semantics would just add the closure containing e_2 instead of the evaluation of e_2.

Example 4.2.6. In the below tree you can follow the semantics of an expression using multiple inference rules. Every horizontal line represents the application of one evaluation rule. ▲

$$\dfrac{\dfrac{\dfrac{}{\{(\texttt{f} \mapsto (\emptyset, \texttt{fix (}\lambda\texttt{f. 4))}\}, \texttt{4} \Downarrow 4}\,(\text{nat})}{\emptyset, \texttt{(fix (}\lambda\texttt{f. 4))} \Downarrow 4}\,(\text{fix}) \qquad 4 > 0 \qquad \dfrac{\dfrac{}{\emptyset, \texttt{2} \Downarrow 2}\,(\text{nat}) \qquad \dfrac{\dfrac{}{\{\texttt{x} \mapsto 2\}, \texttt{x} \Downarrow 2}\,(\text{lookup})}{\{\texttt{x} \mapsto 2\}, \texttt{pred x} \Downarrow 1}\,(\text{pred})}{\emptyset, (\lambda\texttt{x. pred x})\ \texttt{2} \Downarrow 1}\,(\text{app})}{\emptyset, \texttt{ifz (fix (}\lambda\texttt{f. 4)) then 3 else (}\lambda\texttt{x. pred x) 2} \Downarrow 1}\,(\text{if-else})$$

4.3 Answer Set Programming

Answer Set Programming [15] (ASP) is a modelling language with a strong basis in logic programming. It is mainly used as a language to specify NP-hard search problems [99]. There are a lot of different systems supporting a unified ASP standard [17]. An ASP program is essentially a logic program with some extra syntactic restrictions. An ASP solver computes the answer sets of the program under the stable semantics. An answer set consists of a set of atoms which together represent the solution of a problem. One program may have zero, one or multiple answer sets.

4.3.1 Language

An ASP program is a set of rules of the form:

$$\texttt{head :- body}_1\texttt{,...,body}_n\texttt{,not body}_{n+1}\texttt{,...,not body}_m$$

The first n body atoms are positive, the others are negative. The head and body atoms of the rules are of the form $\texttt{id(term}_1\texttt{,...,term}_n\texttt{)}$. Body atoms can also be comparisons ($<$,$>$,$=$,$\neq$) between terms. Terms can be either constants, variables, or arithmetic expressions over terms. Constants are numbers, named constants (strings starting with a lowercase character) or constructors (tuples are considered anonymous constructors). Variables are represented as strings starting with an uppercase character. An ASP program is considered *safe* if all rules are safe. A rule is considered safe if all variables occurring in the rule, occur at least once in a positive body. If the head is omitted, the rule is considered a constraint. In this case no instantations of the body of the rule should exist such that all the bodies are true.

Choice rules are a common syntactic extension for ASP. These allow heads of the form $\texttt{c}_l$ `{a(X) : b(X) }` $\texttt{c}_u$, where $c_l, c_u \in \mathbb{N}$ and $c_l \leq c_u$. This head is considered true if between c_l and c_u instances of `a(X)` are true, given `b(X)`. They allow to easily introduce symbols that are not uniquely defined. We can for instance declare p to be a singleton containing a number between 1 and 10 with

Listing 4.1: An example ASP program and its solutions

(a) An ASP Program

```
1 p(1). p(2). p(3). p(4).
2 1 {q(X) : p(X) } 2.
3 r(X + Y) :- q(X), q(Y).
4 :- not r(5).
```

(b) The Answer Sets

```
Answer Set 1:
p(1) p(2) p(3) p(4)
q(1) q(4) r(2) r(5) r(8)
Answer Set 2:
p(1) p(2) p(3) p(4)
q(2) q(3) r(4) r(5) r(6)
```

the choice rule: `1 {p(X) : X = 1..10 } 1`. The ASP program containing only this line has 10 answer sets, one for each possible singleton.

Example 4.3.1. In Listing 4.1 you can see an example ASP program together with its answer sets. The first line of the program defines the predicate p as the numbers between 1 and 4. The second line is a choice rule with no bodies. It states that q is a subset of p and contains 1 or 2 elements. The third line says that r is the sum of any two elements (possibly the same one) from q. The fourth line asserts that r should contain 5. ▲

4.3.2 Grounding (and Solving)

To understand the details of the translation mechanism, basic knowledge of how an ASP system constructs an answer set is needed. Constructing answer sets happens in two phases: grounding and solving [79]. The grounding process transforms the ASP program to an equivalent propositional program. The solver then constructs the actual answer sets from this propositional format. The translation from PCF described in this chapter will produce a fully positive, monotone theory without choice rules or constraints, i.e., there are no negations present in the program. ASP grounders produce the actual (unique) answer set for this kind of programs. Note that not all ASP systems use the same algorithms, but the information presented here is common to most systems.

The grounding process uses a bottom-up induction of the program. At any point in time, the grounder contains a set of atoms which are possibly part of an answer set. This set starts empty, and by the end of the process this set contains an overapproximation of all answer sets. The grounder tries to instantiate rules using this set of atoms. Whenever a rule is instantiated, the instantiated head is added to this set, and the ground instantiation of the rule is added to the grounding of the program. ASP grounders require that all variables occur in a positive body atom, this is the so-called safety requirement on rules. Safe rules

have the property that only the positive part of the program is essential for finding all rule instantiations and current grounding approaches heavily rely on this property.

Example 4.3.2. Consider the rule `d(X-1) :- d(X), X > 0` and the current set of grounded atoms is just the singleton {`d(1)`}. The grounder can now instantiate the body atom `d(X)` with `X`= 1. The other body atom $1 > 0$ can be statically evaluated to be true. This leads to the newly ground rule `d(0) :- d(1)` and `d(0)` is added to the set of grounded atoms. The grounder can now try to instantiate the rule with `X`= 0, but the comparison $0 > 0$ prevents the rule from being added to the ground program. ▲

Note that the restriction of safe rules do not guarantee that the grounding itself is finite. Through the use of constructors or arithmetic it is still possible to ground infinitely many atoms.

Example 4.3.3. If we just omit `X > 0` from our previous example. The grounder would not stop at `0` and the grounder would keep deriving newly ground rules for smaller and smaller numbers. ▲

After the grounding phase, an ASP solver can produce the actual answer sets based on the grounding. An ASP solver typically uses a SAT solver extended with some ASP specific propagators. The inner workings of these programs are not needed to understand the contents of this chapter.

4.4 Translation

In this section we explain the core of the translation mechanism. Section 4.4.1 defines the relation between the translation and the PCF semantics. Section 4.4.2 introduces some conventions which explain the structure of the resulting program. Finally, Section 4.4.3 explains the static part of the translation. Section 4.4.4 defines the translation relation between PCF expressions and the dynamic part of the translation.

4.4.1 Characterisation of the translation

The semantics of PCF are defined through an inductive definition and ASP supports inductive definitions. An intuitive way of supporting PCF in ASP would be to translate the full evaluation relation to an ASP specification. A PCF expression would then directly correspond to a term in ASP. Unfortunately this is not possible.

Environments are represented by a function from identifiers to closures. And these environments themselves are part of the evaluation relation. This shows that the evaluation relation is a higher-order concept which can not be represented directly in ASP. It is however possible to have a more specialized translation so that the higher-order aspects are removed when translating a particular expression.

Translation relation

The translation is characterised using a relation $\rightsquigarrow$ which we will write as follows:

$$(\mathrm{E},S_1),\ e\ \rightsquigarrow A,\ (\mathrm{t},S_2).$$

E is a mapping from PCF-variables to ASP-terms for at least the free variables in e. This works analogously to the environment of the PCF semantics, which was the mapping from PCF-variables to closures.

S_1 is a set of ASP atoms ensuring the ASP-terms in E are safe and constraints enforcing the ifzero-semantics.

e is the PCF expression that is translated.

A is the ASP program consisting of a set of safe ASP rules, this is the program that contains all the helper rules to translate e.

t is the ASP term which represents the translation of e.

S_2 is the set of ASP atoms ensuring that t is safe.

It can be unintuitive that there are ASP-terms occuring on both sides of the translation relation. The explanation for this lies in the handling of free variables. The translation relation will be defined structurally, this means that for the translation of a composite term, the translation of its subterms are needed. This implies that when translating the expression `(λx. x)`, the subterm `x` needs to be translated as well. The translation needs some context to interpret this `x` and the context of a translation environment will be some information about the parts which are already translated.

A PCF expression corresponds to a single value, but a logic program corresponds to an answer set with a lot of atoms. We need a way to indicate the actual value that is meant with the logic program. The *result*-predicate is used to indicate the resulting value of the program.

Definition 4.4.1. The ASP translation of a PCF expression e determined by $\rightsquigarrow$ is the ASP program A such that $(\emptyset, \emptyset), e \rightsquigarrow A', (t, S)$ and $A = A' \cup \{$`result(t):-S`$\}$.

Soundness of the translation

PCF inherently works on expressions which evaluate to a particular value, ASP programs define relations. A certain equivalence criterion is needed to validate the translation. For this we use the *result*-predicate. For ease of defining the correspondence the soundness criterion is restricted to programs with a numeric evaluation.

Definition 4.4.2. A *sound translator* for PCF to ASP maps every PCF expression e to an ASP program A with a unique answer set. This answer set contains at most one atom for the *result*-predicate. If $\emptyset, e \Downarrow n \in \mathbb{N}$, then $result(n)$ must be an element of the answer set of A.

In the following part of this chapter we will define $\rightsquigarrow$ so that it will be a sound translator. Some intuitions about proving this will be given in Section 4.4.5

4.4.2 Conventions

In the translation, all PCF expressions e correspond to a tuple (t, S) where t is an ASP term and S is a set of ASP bodies. Natural numbers have constants in both PCF and ASP which have a natural correspondence. PCF functions are identified by an ASP term t_f, so that for every ASP term t_x, the tuple $(Y, \{inter((t_f, t_x), Y)\})$ denotes the image of the function t_f applied to t_x. All PCF functions have infinite domains, and thus the full function cannot be represented in a finite answer set. The *domain* predicate serves the purpose of making a finite estimate of the relevant parts of the function. If at some point in the evaluation of e, the function t_f is applied to the value t_x, $domain(t_f, t_x)$ should be true. The *inter*-predicate only needs to be defined for the domain of the function, resulting in a finite answer set containing the relevant parts of the interpretation of the function.

Remember that *result* predicate is used as the predicate determining the final result of the program. So the translation of a PCF expression is an ASP program, defining only 3 predicates:

1. *inter*: determines the interpretation of functions. An atom `inter((F,X),Y)` can be informally read as, the function `F` applied to the argument `X` evaluates to `Y`.

2. *domain*: determines the (relevant) domain of functions. An atom `domain(F,X)` can be informally read as, the function `F` needs to be evaluated for the argument `X`.

3. *result*: determines the end result. It is a unary predicate and should only contain one atom `result(X)` so that `X` is the evaluation of the expression.

Magic Set Transformation

In these conventions a link with the magic set transformations [93] of logic programs appears. The magic set transformation allows us to transform a query, which is traditionally executed top-down, to a program, which can be executed bottom-up. It uses the *magic* predicates to indicate which subqueries needs to be performed. As explained in Section 4.3.2, ASP uses a bottom-up grounding process. So, the translation from PCF to ASP also converts a top-down query (the evaluation of PCF) to a bottom-up process (the ASP grounding). The *domain*-predicate has a function similar to the *magic* predicates: it indicates for which arguments a function needs to be calculated.

4.4.3 Static Preamble

The translation of any PCF expression consists of a dynamic part and a static part. The static part ensures that the interpetation of the `succ`, `pred` and `fix` builtins is taken care of. The dynamic part is produced by the translation algorithm and takes care of the actual PCF expression. The static part is the same for every translation and can be seen in Listing 4.2. The first two lines of the static part ensure the right translation of the `pred` and `succ` terms. E.g., the PCF-term `succ` correctly corresponds to the ASP tuple $(succ, \{\})$ according to the conventions defined in Section 4.4.2. For instance, if somewhere the PCF-term `succ 0` needs to be evaluated. The term will translate to $(Y, \{inter((succ, 0), Y)\})$ which will result in Y being equal to 1 in the answer set.

Just like `pred` and `succ`, the PCF- and ASP-term of `fix` are the same. But the required rules in the preamble are more complex. A naive translation could look like this:

```
inter((fix,F),Z) :- inter((fix,F),Y), inter((F,Y),Z).
```

This rule syntactically represents `fix` f `=` f `(fix` f`)`, but in the stable semantics this equation is not correctly represented by the above rule. Instead, an intermediate term `fixed(F)` is introduced to symbolically represent the

Listing 4.2: Static preamble of the ASP translation

```
1 inter((pred,X),X-1)   :- domain(pred,X), X > 0.
2 inter((succ,X),X+1)   :- domain(succ,X).
3 inter((fix,F),Y)      :- domain(fix,F), inter((F,fixed(F)),Y).
4 inter((fixed(F),X),Y) :- domain(fixed(F),X),
5                          inter((F,fixed(F)),FIX), inter((FIX,X),Y).
6 domain(F,fixed(F))    :- domain(fix,F).
7 domain(FIX,X)   :- domain(fixed(F),X), inter((F,fixed(F)),FIX).
```

fixpoint of `F` in ASP. Now we are able to write the fixpoint as the function `F` applied to the symbolic function `fixed(F)` as can be seen on line 3. This is a new constant, which will identify the fixpoint of F. If the fixpoint is a function, we need to be able to apply it to arguments. Line 4 serves this purpose: to apply `X` to a fixpoint of a function, you can apply `F` to this fixpoint (to ensure we do not have the symbolic representation) and then apply `X` to the result. Finally, lines 6 and 7 ensure that the function applications performed in lines 3 and 4 are all well-defined through the domain predicates.

4.4.4 Translation Algorithm

In this section we present the translation algorithm as a definition for the translation relation $\rightsquigarrow$ using inference rules. Sometimes new ASP constants or variables are needed in the translation. We suppose there is some global supply of those. We use the notation $head \leftarrow B$ for the ASP rule where $head$ is the head atom and B is the set of body atoms.

Scoping

When translating an expression, the free variables in this expression need to be filled in. As we translate nested expressions level per level, we need to pass these values along the expression tree. For this reason, we do not just associate an identifier with a function but a tuple containing an identifier and the current scope. The current scope is a tupling of the full codomain of the translation environment E. We will refer to it as $scope_E$.

Builtins (numbers, pred, succ, fix)

$$\frac{}{\text{(E,S)},\ b\ \rightsquigarrow \emptyset,\ (b,\text{S})}$$

Builtins are relatively easy to translate. The work is taken care of by the static preamble described in Section 4.4.3. A builtin produces no new ASP rules and is translated by itself. Safety is however taken into account, not for the scoping of variables, but for the handling of the if-zero constraints.

Variable

$$\frac{}{\text{(E,S)},\ \mathsf{x}\ \rightsquigarrow \emptyset,\ \text{(E[x],S)}}$$

The algorithm carries around a mapping that represents how variables should be translated. This makes translating a simple variable easy: just look it up in the mapping and combine it with the required safety.

Application

$$\frac{\text{(E,S)},\ e_1\ \rightsquigarrow A_1,\ (t_1, B_1) \qquad \text{(E,S)},\ e_2\ \rightsquigarrow A_2,\ (t_2, B_2)}{\text{(E,S)},\ e_1 e_2\ \rightsquigarrow A_1 \cup A_2 \cup \mathit{rule}_{\mathit{domain}},\ (X, \mathit{body}_{\mathit{inter}} \cup B_1 \cup B_2)}$$

$$X = \textit{a new ASP variable}$$

$$\mathit{rule}_{\mathit{domain}} = \{\mathit{domain}(t_1, t_2) \leftarrow B_1 \cup B_2\}$$

$$\mathit{body}_{\mathit{inter}} = \{\mathit{inter}((t_1, t_2), X)\}$$

Applications are translated by independently translating the two subexpressions. The produced ASP programs need to be combined, with the additional rule that t_2 should be added to the domain of the function t_1. To obtain the resulting value, we use the *inter*-predicate according to the conventions explained in Section 4.4.2.

Example 4.4.3. The rule below shows how the application rule can be used to translate the successor of 1. The static part of the translation ensures that the `inter`-relation for `succ` is interpreted correctly so that in any solution the X gets evaluated to 2.

$$\frac{(\emptyset,\emptyset),\ \texttt{succ}\ \rightsquigarrow \emptyset,\ (\texttt{succ},\emptyset) \qquad (\emptyset,\emptyset),\ 1\ \rightsquigarrow \emptyset,\ (1,\emptyset)}{(\emptyset,\emptyset),\ \texttt{succ 1}\ \rightsquigarrow \{domain(succ,1)\},\ (X, inter((succ,1),X))}$$

▲

Lambda

$$\frac{(\mathrm{E} \cup (\texttt{x},X),S \cup body_{domain}),\ e\ \rightsquigarrow A,\ (t,B)}{(\mathrm{E},\mathrm{S}),\ \lambda\texttt{x. e}\ \rightsquigarrow A \cup rule_{inter}\ ,\ ((l,scope_E),S)}$$

$$X = a\ new\ ASP\ variable$$

$$l = a\ new\ ASP\ constant$$

$$rule_{inter} = \{inter(((l,scope_E),X),t) \leftarrow B\}$$

$$body_{domain} = \{domain((l,scope_E),X)\}$$

Lambda expressions bring a new variable into scope, so they modify the (E,S)-environment before recursively translating the body of the expression. The freshly scoped variable needs to be put into the scoping function E, for this we assign it a new ASP variable (X in the rule). This variable should have a finite range, so we invent a new name for our function (l in the rule) and we use the *domain* predicate to restrict X to the domain of the function. The resulting translation (t,B) represents the image of the function, so the rule $rule_{inter}$ is added to couple the representation of the function to its interpretation.

Example 4.4.4. Consider the following element of the translation relation.
$(\emptyset,\emptyset)(\lambda\texttt{x.2}) \rightsquigarrow \{inter(((l,()),X),2) \leftarrow domain((l,()),X)\},((l,()),\emptyset)$
This can be read as follows: The translation of the constant function to 2 in an empty environment is represented by the constant $(l,())$. The interpretation of $(l,())$ when applied to any term X in the domain of $(l,())$ is 2. ▲

Example 4.4.5. When translating nested lambda expressions, we need the scoping tuple to ensure that all abstracted variables are handled properly. There are two lines of the translation for the expression `(λx. (λy. x))`.

```
inter(((l0,()),X0),(l1,(X0))):-domain((l0,()),X0).
inter(((l1,(X0)),X1),X0)    :-domain((l0,()),X0),domain((l1,(X0)),X1).
```

The constant `l0` corresponds to the outer expressions and the constant `l1` corresponds to the inner expression. The ASP-variable `X0` corresponds to the PCF-variable `x` and the ASP-variable `X1` corresponds to the PCF-variable `y`. ▲

If zero-then-else

$$\frac{\begin{array}{c}(E,S),\ \mathtt{e_{ifz}} \rightsquigarrow A_{\mathtt{ifz}},\ (t_{\mathtt{ifz}}, B_{\mathtt{ifz}}) \\ (E, B_{\mathtt{ifz}} \cup \{t_{\mathtt{ifz}} = 0\}),\ \mathtt{e_{then}} \rightsquigarrow A_{\mathtt{then}},\ (t_{\mathtt{then}}, B_{\mathtt{then}}) \\ (E, B_{\mathtt{ifz}} \cup \{t_{\mathtt{ifz}} \neq 0\}),\ \mathtt{e_{else}} \rightsquigarrow A_{\mathtt{else}},\ (t_{\mathtt{else}}, B_{\mathtt{else}})\end{array}}{(E,S),\ \mathtt{if\ e_{ifz}\ \ then\ e_{then}\ \ else\ e_{else}} \rightsquigarrow A_{\mathtt{ite}} \cup\ rule_{\mathtt{ite}}, (X, S \cup body_{\mathtt{ite}})}$$

$$\begin{aligned} X =& \textit{a new ASP variable} \\ ite =& \textit{a new ASP constant} \\ rule_{ite} =& \{inter((ite, scope_E), t_{\mathtt{then}}) \leftarrow B_{\mathtt{then}} \\ & \quad , inter((ite, scope_E), t_{\mathtt{else}}) \leftarrow B_{\mathtt{else}}\} \\ body_{ite} =& \{inter((ite, scope_E), X)\} \\ A_{\mathtt{ite}} =& A_{\mathtt{ifz}} \cup A_{\mathtt{then}} \cup A_{\mathtt{else}} \end{aligned}$$

If zero expressions are translated using the translations of its three subexpressions. But we need to alter the safety to ensure that the "then"-part is only evaluated if the "if"-part is 0 (and the analog for the "else" part). To construct the value of the full expression we define an intermediate symbol (*ite* in the rule) to represent the union of the "then" and the "else" part. Because the extra safety ($= 0$, $\neq 0$) is mutually exclusive, only one of those terms will have a denotation, so the interpretation of *ite* will be unique.

Example 4.4.6. The translation of `(λx. ifz x then succ else pred) 2 4` is visible in Listing 4.3. The static part is omitted. Lines 1 and 2 are result of the if zero-then-else translation. Line 3 is the result of the lambda translation. Lines 4 and 5 are the result of the application. And in line 6 the end result can be seen. This rule can be read as follows: Let X2 be the application of the function to 2. Let X3 be application of X2 to 4. The final result is X3. ▲

Listing 4.3: Translation of (λx. ifz x then succ else pred) 2 4

```
1 inter((ite1,(X0)),succ):-domain((l0,()),X0),X0=0.
2 inter((ite1,(X0)),pred):-domain((l0,()),X0),X0<>0.
3 inter(((l0,()),X0),X1):-domain((l0,()),X0),inter((ite1,(X0)),X1
      ).
4 domain((l0,()),2).
5 domain(X2,4):-inter(((l0,()),2),X2).
6 result(X3):-inter(((l0,()),2),X2),inter((X2,4),X3).
7 % omitted static part visible in Listing 1
```

4.4.5 Soundness

Section 4.4.1 claims that the translation relation produces a sound translation of PCF to ASP. A sketch for proving this soundness is given here by the use of induction.

Theorem 4.4.7. *The translation of PCF expressions determined by* $\rightsquigarrow$ *is a sound translator.*

Proof. (sketch)

Expressions consisting of simple *numbers* trivially satisfy the theorem. These expressions have a translation which directly corresponds to `result(n)` where `n` is the number in the expression.

For expressions which are *applications* of the form `f x`, the result will look like `result(R) :- inter((f,x),R)`. We can suppose through the induction hypothesis that `f` and `x` are properly translated, we need to show that the rule with head `inter((f,x),R)` has a true instantiation for the body. This falls under our induction hypothesis except for the domain-restriction in the function. However, the translation simply adds the `x` to the domain of `f` so this is also fullfilled.

If zero-then-else expressions of the form `ifz i then t else e` have two main rules. One rule which for when the `then`-part is the value of the expression and one for the `else`-part. The result will look like `result(R) :- inter(ite,R)`. Suppose that the expression `i` evaluates to zero. Then the rule responsible for the then-part will be equal to the translation of the then-part by itself as the extra safety condition is trivially true (this corresponds to $t_{\texttt{ifz}}$ `= 0` in the set of translation rules). The complete analog is true for the else-part of the expression.

Lambda expressions and *builtin functions* can not evaluate to a number by themselves so these expressions can never lead to an unsound translation. □

4.4.6 Implementation

An implementation was made in Kotlin. The runtime uses Clingo [59] to run the resulting ASP files, but the resulting specifications could be used with any ASP-Core-2 [17] compliant system. On `https://dtai.cs.kuleuven.be/krr/pcf2asp` you can find a tool on which you can try out the translation. A few example PCF formulas are provided, but you can ask for translations of arbitrary PCF formulas and see their corresponding answer set.

4.5 Applications

4.5.1 Multiple interpretations for one variable

Directly translating PCF gives us little more than a traditional interpreter of PCF would do, but based on this translation we can provide extra functionality, leveraging the existing ASP solvers. Traditional PCF does not support the possibility that the interpretation of a term is not uniquely defined, but we can extend PCF so we can declare the variable a as a number between 1 and 10 without defining its specific value. In that case we can get (at most) 10 different evaluations of our program, one for each interpretation of a. It is easy to extend the translation to encode this in ASP.

Traditional interpreters solve the question: "What is the evaluation of this program?". But using these variables another question can be interesting: what value(s) for a should I choose so that the program evaluates to 0. We can leverage the strengths of ASP solvers to find the solutions. Expressing that the evaluation should be zero can be done through a simple ASP constraint:

```
:- not result(0).
```

When this constraint is added, the resulting answer sets will now all have the same interpretation (0) for the *result* predicate, but we are interested in the interpretation for a.

Example 4.5.1. In Listing 4.4 you can see a PCF expression representing that $a + b = c$. If we now use choice rules in ASP to translate these variables to the domain of natural numbers between 0 and 10, we can use ASP to find multiple

Listing 4.4: $a + b = c$ in PCF

```
(λeq. λplus.
      eq (plus a b) c)

(fix (λeq.λx.λy. ifz x then (ifz y then 0 else 1)
      else (ifz y then 1 else eq (pred x) (pred y))))
(fix (λplus.λx.λy. ifz y then x
      else plus (succ x) (pred y)))
```

Listing 4.5: $a + b = c$ in ASP

```
1{a(X) : X=1..10}1.
1{b(X) : X=1..10}1.
1{c(X) : X=1..10}1.
:- not result(0).
...
domain(X1,A):-domain((l0,()),X0),domain((l1,(X0)),X1),a(A).
...
```

solutions of this equation. An example of how this would look in ASP an be seen in Listing 4.5. ▲

The problem in Example 4.5.1 can easily be generalised to arbitrarily complex polynomials to model mixed integer problems. A graph coloring problem can be represented by using a new constant for each node that needs to be colored and writing down an expression that evaluated to 0 if the graph is colored correctly.

An important thing to note here is that ASP does not naively calculate the result for all possible values of the choice rules. Instead, it uses a CDCL-based solving algorithm to explore the search space in an intelligent way.

4.6 Towards a more expressive language

PCF is not intended to be an end-user language, but it serves as a basis for many real world programming languages. Analogously, we are developing a more expressive language based on the principles of PCF. This language includes more complex data types for representations which are more elegant than possible

in PCF. Together with the multiple-model semantics of ASP this leads to an interesting modelling language.

Using these ideas the Functional Modelling System (FMS) is introduced in Chapter 5. This system is an extension of PCF with some more practical language constructs and uses the translation principles described in this chapter to use ASP as a solver engine for this new language. However as indicated in Section 4.6.5, a lot of optimisations are needed to be competitive with native ASP encodings. The efficiency of these translators have not been formally investigated yet.

4.6.1 Builtin Functions

It is easy to extend the evaluation and the translation relation with additional builtin functions. For instance, basic arithmetic operations can be added to the language. This does not increase the theoretical expressivity of the language, as these functions can already be defined through PCF (cf. Listing 4.4). However, it does increase the practical usability and execution speed of the language.

Example 4.6.1. If we allow arithmetic expressions to occur in PCF expressions. We can define the semantics of the extended language by adding one rule to the semantics of the language. Analogously, we can define the translation relation of the extended language by adding one rule to the translation relation of the original language.

$$\frac{E,\ \mathtt{e}_1\ \Downarrow n_1 \qquad E,\ \mathtt{e}_2\ \Downarrow n_2}{E,\ \mathtt{e}_1\ \ \mathtt{+}\ \mathtt{e}_2\ \Downarrow n_1 + n_2}$$

$$\frac{\begin{array}{c}(E,S),\ \mathtt{e}_1\ \rightsquigarrow A_1,\ (t_1, B_1)\\ (E,S),\ \mathtt{e}_2\ \rightsquigarrow A_2,\ (t_2, B_2)\end{array}}{(E,S),\ \mathtt{e}_1\ \ \mathtt{+}\ \mathtt{e}_2\ \rightsquigarrow A_1 \cup\ A_2, (t_1 + t_2, B_1 \cup\ B_2)}$$

▲

This example is trivially extended to all functions which are directly available as terms in Answer Set Programming. This includes the basic arithmetic operators, modulo operations and absolute value operators.

When a particular function is not available inside of ASP, it is often possible to exploit a common extension of ASP for calling external functions[49]. ASP

implementations such as Clingo have support for Python and Lua scripts. These can be used to couple functions to a procedural interpretation.

All functions operating on integers can be added to the language with this mechanism. However, these external function calls have their own limitations. These functions work on ASP-terms and thus can not easily handle the ASP encoding we chose for functions. So higher-order functions are not easily injected in ASP through this mechanism.

4.6.2 Let expression

Let expressions are a language construct which is available in a lot of languages. These expressions allow to give a local definition to a symbol. The main advantage of having these in the language is readability as there are easy transformations to eliminate them from any expression. There are a few different possible variants of this language construct, depending on the amount of symbols which are defined and whether recursion is allowed.

Single Non-recursive Definition In the case of non-recursive definitions, the defined symbol can simply be evaluated and given to the inner expression. This corresponds exactly to the semantics of the function application. So we can use function application to define the semantics of the let expression:

$$\frac{E, (\lambda \texttt{x. f}) \texttt{ e} \Downarrow v}{E, \texttt{let x = e in f} \Downarrow v}$$

As this is a simple conversion which never introduces new let expressions, we can just translate a let expressions as if the function application was in its place.

$$\frac{(E, S), (\lambda \texttt{x. f}) \texttt{ e} \rightsquigarrow A, (t, B)}{(E, S), \texttt{let x = e in f} \rightsquigarrow A, (t, B)}$$

Single Recursive Definition The recursive version of the let expression allows to use the defined symbol in its definition. This version of the let expression would allow the first proposed version of the doubling function from Example 4.2.2.

$$\frac{E, (\lambda \texttt{x. f}) \texttt{ (fix } (\lambda \texttt{x. e)) } \Downarrow v}{E, \texttt{letrec x = e in f} \Downarrow v}$$

For the translation relation, we use the same reasoning as for the non-recursive case and just transform the input expression to its counterpart without letrec to translate it.

$$\frac{(E,S), (\lambda \texttt{x. f}) \ (\texttt{fix} \ (\lambda \texttt{x. e})) \ \rightsquigarrow A, (t,B)}{(E,S), \ \texttt{letrec x = e in f} \ \rightsquigarrow A, (t,B)}$$

Multiple Mutually Recursive Definitions The last version of the let expression is the most advanced one. It also allows to define symbols by mutual recursion. This version strictly generalises the single recursive case so we will use the same syntax for it.

Example 4.6.2. A traditional example of mutual recursive definitions are the definitions for odd and even. We will use 0 and 1 to respectively mean true and false. A number is considered even if its predecessor is odd and vice versa. The following expression would evaluate to `0`.

```
letrec even x = ifz x then 0 else odd (pred x)
       odd  x = ifz x then 1 else even (pred x)
 in odd 2019
```

▲

A formal handling of this version of the language construct requires a polyvariadic version of the fixpoint combinator [1]. The formal specification of Scheme[80] (a functional language including this form of letrec) uses side effects to transform this language construct away. A proper handling of this language construct while keeping the formalisation pure looks a bit more convoluted. This would take up too much space in this chapter and for this, we refer to the Appendix A.

4.6.3 Builtin Finite Sets

The semantics of Answer Set Programming declare sets as the interpretation of predicate symbols, so sets are easily encoded in ASP. It is interesting to add these sets as a data primitive to PCF.

PCF Semantics To add sets to PCF we will need to extend the set of values in its semantics. Apart from a closure or natural number, a value can now also be a set of other values. Using this modified definition, we can add the new language constructs and their semantics.

We extend PCF with terms denoting finite sets. The basic set term is an expression of the form $\{\mathtt{e}_1 \ \ldots \ \mathtt{e}_n\}$. The following rule determines the semantics of these expressions.

$$\frac{\mathrm{E}, \mathtt{e}_1 \Downarrow v_1 \qquad \ldots \qquad \mathrm{E}, \mathtt{e}_n \Downarrow v_n}{\mathrm{E}, \{\mathtt{e}_1 \ \ldots \ \mathtt{e}_n\} \Downarrow \{v_1 \ldots v_n\}}$$

Additionally we add some functions to transform sets: `map` (pointwise function application), `filter` (only keep elements according to a certain criterion) and `concat` (flattening a set of sets to a single set). Because we haven't added booleans to the language yet we will filter the elements so that we keep the ones for which a particular functions evaluates to zero (instead of the more traditional evaluation to True). The last function we add is the `size` function which maps a set to the number of elements it contains.

$$\frac{\mathrm{E}, \mathtt{s} \Downarrow \{\mathtt{v}_1 \ldots \mathtt{v}_n\} \qquad \mathrm{E}, \mathtt{f}\ \mathtt{v}_1 \Downarrow w_1 \qquad \ldots \qquad \mathrm{E}, \mathtt{f}\ \mathtt{v}_n \Downarrow w_n}{\mathrm{E}, \mathtt{map}\ \mathtt{f}\ \mathtt{s} \Downarrow \{w_1 \ldots w_n\}}$$

$$\frac{\mathrm{E}, \mathtt{s} \Downarrow \{\mathtt{v}_1 \ldots \mathtt{v}_n\} \qquad \mathrm{E}, \mathtt{f}\ \mathtt{v}_1 \Downarrow b_1 \qquad \ldots \qquad \mathrm{E}, \mathtt{f}\ \mathtt{v}_n \Downarrow b_n}{\mathrm{E}, \mathtt{filter}\ \mathtt{f}\ \mathtt{s} \Downarrow \{w_i | i \in 1 \ldots n, b_i = 0\}}$$

$$\frac{\mathrm{E}, \mathtt{s} \Downarrow \{\mathtt{v}_1 \ldots \mathtt{v}_n\} \qquad \mathrm{E}, \mathtt{f}\ \mathtt{v}_1 \Downarrow w_1 \qquad \ldots \qquad \mathrm{E}, \mathtt{f}\ \mathtt{v}_n \Downarrow w_n}{\mathrm{E}, \mathtt{concat}\ \mathtt{s} \Downarrow \{a | i \in 1 \ldots n, a \in w_i\}}$$

$$\frac{\mathrm{E}, \mathtt{s} \Downarrow \{\mathtt{v}_1 \ldots \mathtt{v}_n\}}{\mathrm{E}, \mathtt{size}\ \mathtt{s} \Downarrow n}$$

ASP Translation To encode sets in ASP we will add one new predicate to the conventions made in Section 4.4.2. The *member* predicate will encode the membership relation of sets. Sets in the extended PCF language are identified by an ASP term t_S, so that the object identified by an ASP term t_x is in the set iff $member(t_S, t_x)$ is true.

The last step is extending the translation relation with the new language constructs. The set construction construct has a straightforward direct translation: the translation of the elements gets added in the *member*-relation to a new identifier representing the set.

$$\frac{\forall i \in 1..n : (\text{E,S}),\ e_i \ \rightsquigarrow A_i,\ (t_i, B_i)}{(\text{E,S}), \texttt{\{e}_1\texttt{...e}_n\texttt{\}} \ \rightsquigarrow (\bigcup A_i) \cup rules_{member}\ ,\ ((l, scope_E), S)}$$

$$l = \textit{a new ASP constant}$$

$$rules_{member} = \{member(((l, scope_E), t_i)) \leftarrow B_i | i \in 1..n\}$$

The other new functions can simply be translated to a constant with the same name and their interpretation can be added to the preamble. The extension to the preamble can be found in Listing 4.6. The definitions are more complex than the original ones, but only consist of manipulating the *member*,*inter* and *domain*-predicates according to their conventions.

Common Set Operations We can easily combine the available set operations to obtain other common operations on sets. For practical purposes, it can be useful to have such functions available as a library. Importing a library can be modelled as adding a big let expression around the other expressions. This let expression contains all symbols defined in the library.

Example 4.6.3. Checking if some element `e` is member of a set `s` can be written as `member s x = size (filter (λx. x == e) s) == 1`
The union of two sets `s1` and `s2` can be written as
`union s1 s2 = concat {s1,s2}`
The intersection of two sets `s1` and `s2` can be written as
`intersect s1 s2 = filter (member s2) s1`
Statement `f` should hold for all elements of a set `s` can be written as
`forall s f = size s == size (filter f s)` ▲

4.6.4 Tupling

Tuples enable a programmer to encode more advanced data types. Answer Set Programming has native support for tuples, so adding them to the translation

Listing 4.6: Extension of the Static Preamble for Sets

```
% Definition of map
inter((map,F),map(F)) :- domain(map,F).
inter((map(F),S),map(F,S)) :-
    domain(map,F), domain(map(F),S).
member(map(F,S),X) :- member(S,Y),
    inter((F,Y),X), domain(map,F), domain(map(F),S).
domain(F,X) :- domain(map(F),S), member(S,X).

% Definition of filter
inter((filter,F),filter(F)) :- domain(filter,F).
inter((filter(F),S),filter(F,S)) :-
    domain(filter,F), domain(filter(F),S).
member(filter(F,S),Y) :-
    member(S,Y), inter((F,Y),0), domain(filter,F),
    domain(filter(F),S).
domain(F,X) :- domain(filter(F),S), member(S,X).

% Definition of concat
inter((concat,S),concat(S)) :- domain(concat,S).
member(concat(S),Y) :- member(S,X), member(X,Y).

% Definition of size
inter((size,S),N) :- domain(size,S),
    #count{X : member(S,X)} = N.
```

relation is quite straightforward. When a tuple value is obtained, there also needs to be a way to get the different parts of the tuple back out. In ASP this is done through unification. We will add projection function to PCF to obtain this functionality. For simplicity we will only add binary tuples to the language, but this is easily generalised to tuples of any size or even data constructors.

PCF Semantics To add tuples to PCF we will again need to extend the set of values in its semantics. Tuples of values are now also values. Using this modified definition we can simply declare a binary tuple of expression to be an expression and add the projection functions `fst` and `snd` to respectively the first and second element of the tuple.

Listing 4.7: Extension of the Static Preamble for tuples

```
% Definition of fst
inter((fst,(A,B)),A) :- domain(fst,(A,B)).

% Definition of snd
inter((snd,(A,B)),A) :- domain(fst,(A,B)).
```

$$\frac{\mathrm{E},\ \mathtt{e}_1 \ \Downarrow\ v_1 \qquad \mathrm{E},\ \mathtt{e}_2 \ \Downarrow\ v_2}{\mathrm{E},\ \mathtt{(e_1,\ e_2)} \ \Downarrow\ (v_1, v_2)}$$

$$\frac{\mathrm{E},\ \mathtt{e} \ \Downarrow\ (v_1, v_2)}{\mathrm{E},\ \mathtt{fst\ e} \ \Downarrow\ v_1}$$

$$\frac{\mathrm{E},\ \mathtt{e} \ \Downarrow\ (v_1, v_2)}{\mathrm{E},\ \mathtt{snd\ e} \ \Downarrow\ v_2}$$

ASP Translation For the translation, the tuple construction in PCF corresponds straightforwardly to tuple construction in ASP. Analoguous to the previous section. The projection functions can be translated to a constant with the same name, combined with a preamble extension. This preamble extension can be found in Listing 4.7.

$$\frac{(\mathrm{E,S}),\ e_1 \ \rightsquigarrow\ A_1,\ (t_1, B_1) \qquad (\mathrm{E,S}),\ e_2 \ \rightsquigarrow\ A_2,\ (t_2, B_2)}{(\mathrm{E,S}),\ \mathtt{(}e_1\mathtt{,}e_2\mathtt{)} \ \rightsquigarrow\ A_1 \cup A_2,\ ((t_1, t_2), B_1 \cup B_2)}$$

4.6.5 Optimisations

The translation algorithm for PCF is not an optimal translation. There are multiple ways to increase the performance of the resulting translation. They can be divided into two categories. We can either change the translation algorithm to result in a more performant ASP program, or we can preprocess the input so it is more efficiently handled by the algorithm, which by extension improves the performance of the end result.

Optimising the translation One important optimisation follows from the observation that not all variables in scope need to be present in $scope_E$, only the ones which are actually used in the subexpression. Applying such optimisations can significantly reduce the size of the grounding of the ASP program.

This observation can be further formalised by observing that the following, more general rule for translating lambda-expressions would also have led to a sound translator.

$$\frac{(\mathrm{E} \cup (\mathtt{x}, X), S \cup body_{domain}),\ e \ \rightsquigarrow A,\ (t, B) \qquad \mathrm{E} \cap \mathrm{E}_2 = \emptyset}{(\mathrm{E} \cup \mathrm{E}_2, \mathrm{S}),\ \lambda\mathtt{x.\ e} \ \rightsquigarrow A \cup rule_{inter}\ ,\ ((l, scope_E), S)}$$

This new rule is strictly more general than the old one, as we can choose E_2 to be empty. However, it also suggests that if we can translate the formula with a reduced E, that the end result will be the same. Restricting E to the free variables in the subexpression `e` will result in smaller scope-tuples. This means that these terms will not evaluated multiple times for environments where no relevant variable has changed. The same reasoning about making environments smaller can also be applied to the other translation rules such as for application.

Example 4.6.4. Consider the following expression: `λx. λy. (λa. x) (succ y)`. The application has 2 variables in scope: `x` and `y`. The right hand does not use `y` and the left hand does not use `x`, so these can be translated using a different environment. `(λa. x)` can be translated using an environment that only contains `x` and `(succ y)` can likewise be translated using an environment that only contains `y`. ▲

This optimisation is not active in the implementation presented in Section 4.4.6 but it is present in the FMS system which will be presented in the next chapter.

Optimising the expression Instead of optimising the translation process, it is also possible to rewrite the input PCF expression to a form which translates to a more efficient ASP program. Techniques of this nature will be explored further in Chapter 5 where the optimisations available in a prototype implementation of a more elaborate language will be explained. Most of those techniques are directly applicable when applied to PCF expressions.

4.7 Typing

Up to this point, we never considered any typing of PCF expressions. For the PCF language (and all extensions that were mentioned in previous sections), typing is completely optional and only serves as a sanity check for expressions.

Enforcing that expressions are well-typed eliminates a lot of expressions from the language. We hope that many unuseful expressions are removed while removing as few as possible from the useful ones.

Definition 4.7.1. A *type* τ is either the basic type `Int` or a function $a \to b$ where a and b are types.

Just like it was the case for the operational semantics, we will also need an environment to evaluate our typing relation in.

Definition 4.7.2. A *typing environment* E is a mapping from identifiers to types.

The typing is characterised using type judgements which we will write as follows: $E \vdash e : \tau$. This can be read as: under the typing environment E, the expression e is typed as τ.

4.7.1 Typing Rules

We again use a similar set of inference rules to define the typing relation.

$$\frac{E(x) = \tau}{\mathrm{E} \vdash x : \tau}$$

$$\frac{\mathrm{E} \vdash e_1 : \tau \to \sigma \qquad \mathrm{E} \vdash e_2 : \tau}{\mathrm{E} \vdash e_1 e_2 : \sigma}$$

$$\frac{\mathrm{E} \cup \{x \to \tau\} \vdash f : \sigma}{\mathrm{E} \vdash \lambda \mathtt{x}.\, f : \tau \to \sigma}$$

$$\frac{n \in \mathbb{N}}{\mathrm{E} \vdash n : \mathtt{Int}}$$

$$\frac{}{\mathrm{E} \vdash \texttt{succ: Int}\rightarrow\texttt{Int}}$$

$$\frac{}{\mathrm{E} \vdash \texttt{pred: Int}\rightarrow\texttt{Int}}$$

$$\frac{\mathrm{E} \vdash e_z : \texttt{Int} \qquad \mathrm{E} \vdash e_t : \tau \qquad \mathrm{E} \vdash e_e : \tau}{\mathrm{E} \vdash \texttt{ifz } e_z \texttt{ then } e_t \texttt{ else } e_e \vdash \tau}$$

$$\frac{}{\mathrm{E} \vdash \texttt{fix: } (\tau \rightarrow \tau) \rightarrow \tau}$$

Definition 4.7.3. We consider an expression e *well-typed* if there is a type τ such that $\emptyset \vdash e : \tau$.

Example 4.7.4. `(λx.x)` is well-typed and has many valid types such as `Int→Int` and `(Int→Int)→(Int→Int)`. ▲

Example 4.7.5. `(λid. id 1) (λx.x)` is well-typed and has one unique type `Int`. ▲

Example 4.7.6. `(λid. (id id) 1) (λx.x)` has no valid typing though the semantics of PCF would associate the value 1 to this expression. ▲

Note that this type system is very crude. It relates multiple types to some expressions and some intuitive expressions are not well-typed. There is also no simple way to implement a type inference algorithm based on these inference rules. Look for instance at the typing rule for abstraction. To mechanise this rule into an algorithm, we need to invent a τ for the environment so that the type of f can be derived.

4.7.2 Extensions

There are many extensions and variations to the set of inference rules. The most important one is the addition of polymorphic types. In this polymorphic setting, we can quantify over types so that we can use one function with arguments of different types. Another common variation to these rules introduces the unification of types. The goal of this is to obtain a set of rules which correspond to an algorithm in which no guessing of types is necessary. These strategies are applied by commonly used type inference algorithms.

4.8 Declaring non-defined symbols

In Section 4.6 we put a lot of effort in extensions for PCF. This shows how this base language can be expanded with language constructs to obtain a more usable real-world language. To reach the goal of having a functional modelling language, we need to expand our language with other constructs. We also need a way for declaring non-defined symbols. This enables us to write down a theory with multiple models as described in Section 4.5. These expressions change the nature of the evaluation relation of PCF as these allow for multiple evaluations, depending on the instantiation of this non-defined symbol.

4.8.1 Element of-declarations

PCF Semantics We introduce expressions of the form `let x = element of s in e` where `x` is an identifier, and `s` and `e` are expressions. The informal meaning is that `x` is an element of `s` in the evaluation of the expression `e`. It closely corresponds to Curry's `?` operator, the main difference is that this expression can only occur at the top level. This can be formalised as:

$$\frac{\text{E, } \texttt{s} \Downarrow \{\texttt{v}_1 \ldots \texttt{v}_i\} \qquad i \in \{1..n\} \qquad \text{E} \cup \{(x \mapsto \texttt{v}_n)\}, \texttt{e} \Downarrow v}{\text{E, } \texttt{let x = element of s in e} \Downarrow v}$$

ASP Translation This non-determinism translates to a choice-rule in the resulting ASP program.

$$\frac{\text{(E,S), } \texttt{s} \rightsquigarrow A_1, (t_1, B_1) \qquad (\text{E} \cup \{\texttt{x} \mapsto Y\}, \text{S} \cup \{p(Y)\}), \texttt{e} \rightsquigarrow A_2, (t_2, B_2)}{\text{(E,S), } \texttt{let x = element of s in e} \rightsquigarrow A_1 \cup A_2 \cup rule_{choice}, (t_2, B_2)}$$

$$p = a\ new\ ASP\ predicate$$

$$Y = a\ new\ ASP\ variable$$

$$rule_{choice} = 1\{p(Y) : member(((t_1, scope_E), Y))\}1 \leftarrow B_1$$

This can be read as: translate `s` in the current scope. Use $rule_{choice}$ to define a new predicate containing exactly one element of the set `s`. Then translate the expression `e` while using the only element in this new set as the interpretation for the variable `x`.

4.8.2 Other declarations

The above construction can be generalised for other declarations such as `subset of` a set or even `function between` two sets. The same general principle stays in place, where a choice rule is used to encode the non-deterministic choosing part and the inner expression is subsequently translated without needing to be aware of the non-determinism.

4.9 Conclusion

We presented a translation from PCF to ASP programs. A basic translation is easy to implement, and many optimisations are possible. With only small changes, we can exploit the search power of ASP to solve problems expressed in PCF. This translation can serve as a basis to use functional programming techniques in modelling languages for search problems, or even tighter integrations between functional and logical languages. In the next chapter we explain how these principles were used to implement an actual prototype system: FMS.

5 FMS: Functional Programming as a Modelling Language

A first version of the work of this chapter was presented at the International Workshop on Functional and Logic Programming 2018 [32].

5.1 Introduction

The previous chapter explained a theoretical framework which shows how lambda calculus can be used to define higher order logics. Based on these ideas, this chapter introduces a practical Functional Modelling System (FMS), powered under the hood by an Answer Set Programming (ASP) solver [58]. The system is focused on solving NP-complete search problems, the same class of problems its underlying engine - ASP - can solve. In this chapter we give a high level overview of the system, point out which techniques and algorithms are already implemented and highlight the places where there is low-hanging fruit for improving the system. In Section 5.2 we give an overview of the language of the system, both on the external and the internal level. In Section 5.3 we explain what steps a file goes through to produce an answer. In Section 5.4 we take a deeper look into what kind of optimisation techniques are present in the system. We conclude in Section 5.5 with information on the tooling and availability of the system.

5.1.1 Context

Modelling Languages In this paper we use the concept of *modelling language* to denote a language which is used to express a set of rules to which solutions

should adhere. Typically, modelling languages allow for naturally expressing NP-complete problems. A solution for this set of rules is called a model. Typical examples of such languages are Minizinc[97], ASP[108] and FO(.)[11], the language of IDP3. The new Functional Modelling Language (FML) is an addition to this set of modelling languages. Most previous sytems have their roots in logic programming, and first order logic. The new FML language is based on constructs which are traditionally only found in functional programming languages. With these constructs it is still possible to use traditional first order logic specifications, but there are also new modelling possibilities.

The only other fully higher order modelling language that the authors are aware of is ProB[86]. The expressive power of ProB is very high but the solving capabilities of this language are limited. The system is mainly used to model dynamic systems and not to solve search problems.

Haskell as a host language Monadiccp [110] and Ersatz are Haskell libraries, through which it is also possible to define search problems using a functional language. It allows the user to define constraint satisfaction problems using Haskell, which is then translated to respectively a constraint programming or a SAT solver. While the host language is a fully higher-order language, the problems that can be specified in monadiccp are only first order. Ersatz does have the notion of relations, but somebody writing the specification has to think about the possible range of values while declaring the relation. Oftentimes this is easy, but for example when using a Collatz sequence, it is not easy to tell the range of numbers that will occur in the chain. Ersatz also requires the relation to have a contiguous range as inputs for the relation. This is often natural for numeric problems but this makes modelling less natural when the domain elements have no natural ordering. FMS takes these issues out of the equation as the domain of the function is derived from the specification, so the user does not need to think about the input domain. Ersatz is embedded in Haskell, and this has the consequence that the user has to be conscious whether he is calling native Haskell functions (such as addition) or relations which need to be found by the solver. These have no different syntax in FMS which leads to a more streamlined experience.

5.1.2 Search Technologies

There are a multitude of existing technologies apt for solving search problems. SAT solvers have become very efficient and are the standard for a number of problems, but there are also a lot of other technologies: SMT (SAT modulo theories) which extends SAT with higher level concepts, constraint programming

Listing 5.1: Quantification of functions (in the Full language)

```
c :: element of {3..5}.
d :: element of {3..5}.

s := {\x -> x * 2, \x -> x + c, \x -> x * c}.

! s (\f -> f d < 10).
```

systems, mixed integer programming, ... FMS does not contribute to these search technologies, but to the modelling techniques for these.

For FMS, ASP was chosen. ASP solvers use a SAT solver at their core. These SAT solvers are extended with support for expressions which are difficult (or expensive) to reduce to SAT, such as recursive logic definitions. ASP was chosen because the language itself is already relatively high level, so it is easier to build another language on top of ASP compared to other paradigms such as SAT.

5.1.3 First Order or Higher Order?

FML supports higher order expressions: functions which take functions as input and return new functions. Higher order sets are also supported: it is possible to define `s` as the set $\{\{1,4\},\{1,6,7\},\{5,4,9\}\}$.

We have a polynomial translation to ASP which has first order semantics, from this it follows that the semantics of our language is not higher order. The higher order aspects of the FML language are syntactical sugar for a first order encoding of higher order functionality.

Nevertheless, this approach allows for modelling techniques which were not possible before in modelling languages. For instance in Listing 5.1 you can see an FML specification which declares the constants `c` and `d` as numbers between 3 and 5. The next line defines the set `s` as the set containing the doubling function, the function which adds `c` to a number and the function which multiplies a number by `c`. The last line starts with `!` which means "forall". So the line means: for every function `f` in the set `s`, `f` applied to `d` is smaller than `10`. There is exactly one model for this, which maps both `c` and `d` to 3.

Listing 5.2: Find 2 N-queen solutions which have no shared queens

```
domain := {1..8}.

solution1/1 :: function to domain.
solution2/1 :: function to domain.

alldiff solution f := ! domain (\x ->
                        ! domain (\y -> x ~= y => f x ~= f y)).

nqueens solution := ! {
                        solution,
                        \x -> x - solution x,
                        \x -> x + solution x
                      } (alldiff solution).
nqueens solution1.
nqueens solution2.
! domain (\x -> solution1 x ~= solution2 x).
```

5.1.4 Advantages

There are a number of advantages when comparing FML to existing modelling languages. One driving factor for higher order language is the easy abstraction of concepts. Suppose that you want to find 2 different solutions of the N-queens problem so that no two queens occur in the same place. In traditional ASP or MiniZinc (a constraint programming language), one would need to write down the N-queens constraints twice: once for each of the two solutions. Extensions for ASP have been introduced to introduce templates [18], which circumvent this problem but make a fundamental difference between first order and higher order predicates. FML does not require the user to differentiate between first and higher order terms and allows to easily reuse the higher order predicate `nqueens` indicating that `solution` is a solution to the nqueens problem. The FMS modelling for this problem can be seen in Listing 5.2.

A second advantage of the FML is the propagation of the domains. In most traditional modelling languages, whenever you declare a function or predicate, you need to specify a finite domain. This would disallow a definition of prime numbers as short as "`prime x := ! {2..x-1} (\y -> x % y > 0)`" (x is prime if for all numbers y between 2 and x-1, x modulo y is larger than 0) because this is a function with an infinite domain (all integers). FML allows this by automatically deriving the relevant domain of the function from the constraints.

Listing 5.3: BNF for the Core language

```
e := x                          (variable)
   | e e                        (application)
   | \x -> e                    (lambda abstraction)
   | let (x := e)* in e         (let binding)
   | i                          (injection)
   | case e of (pattern -> e)*  (case expression)
   | b                          (builtin)
   | {e*}                       (set expression)
   | h(e*)                      (herbrand expression)
   | outputexp(s, e)            (output expression)

pattern := _                    (don't-care pattern)
         | h[pattern*]          (constructor match)
         | x                    (variable match)
```

5.2 Language

We differentiate between the *Core* language and the *Full* language of FMS. We start by introducing the Core language in Section 5.2.1. The Core language represents the essentials of FML. The Full language extends this with syntactical sugar for ease of use and is introduced in Section 5.2.2. Internally in the system, all expressions are represented using the Core language. It makes internal transformations easier than a representation that incorporates all details of the Full language.

5.2.1 Core Language

FML is based on the lambda calculus. This lambda calculus is extended with some common constructs (such as let-bindings and numbers) and less common constructs (such as the built-in notion of a set). There are ten language constructs in the Core language. Nine with a semantic meaning, and one to track the values which are needed in the output. This language is only represented by internal data structures inside of the system. It can't be parsed or printed and thus has no explicit syntax. However, it can be useful to see an approximate BNF for this Core language which can be seen in Listing 5.3. The language constructs you can see in the BNF are:

Variable A traditional reference to a bound identifier

Application This construct applies a function to an argument.

Lambda Internally, this is the only way a function can be declared. Lambdas abstract away one single variable. If multiple arguments are required, this language construct should be nested. Note that a tuple counts as a single argument.

Let binding Introduces some local definitions in a new expression. A binding binds a variable to an expression, possibly recursively. This is the only way to bind a term recursively.

Injection This is a way to inject an arbitrary JVM (Java Virtual Machine) object into the language. In the current system, it is only used to inject integers and strings. In the future this construct could be used to reason over arbitrary JVM objects.

Case This language construct handles all pattern matching. This can be seen as a generalised form of an if-then-else construct. An if-then-else construct would only handle patterns: true and false.

Builtin symbol There is quite a list of builtin functions for the FMS. The reason for this is not always that we can't define them in the language, but that they are common functions and a dedicated method for translating them leads to a more efficient translation to ASP. Builtin functions include arithmetic operations (`+`,$*$), boolean functions (`&`,`|`) and set functions (forall: `!`, exists: `?`).

Set Expression Sets could be emulated through the use of constructors but for efficiency reasons we opted to have sets as an explicit built-in datatype of the language. This language construct makes a finite set containing some expressions. To represent all set expressions contained in the Full language, this Core construct needs to be combined with some builtin symbols.

Herbrand Expression This construct corresponds to the declared constructors. It is a named constructor optionally applied to some arguments. These constructors can be pattern matched using the Case construct.

Output Expression This language construct is the only one which says nothing about the actual meaning of the expression but is a construct to annotate that the expression it contains should be outputted using a certain name. `4 + OutputExp("a",5)` would denote an expression that evaluates to 9, but the subexpression `5` should be outputted under the name `"a"`. Most languages use some kind of print statements for this. But as we don't have those in a modelling language like FML, we have to track the values we want to print in another way.

Listing 5.4: Minus One in Peano Arithmetic (in the Full language)

```
s/1 :: constructor.
nil/0 :: constructor.

minusOne x := case x of
    s [ a ] -> a;
    nil []    -> nil;.
```

Types

FML is strongly typed, this means that every expression has a type which is enforced at compile time, before trying to run the program. The language is also implicitly typed, this means that these strong types are not necessary parts of the syntax and it is possible that the input file contains no type annotations.

The type system itself is a Hindley-Milner type system, which includes polymorphism through universal quantification over types. E.g., the forall quantifier has the following type: `∀ a. Set a -> (a -> Bool) -> Bool`. In the implementation itself, the universal quantification over types is left implicit. You can see here that the type system also enforces that all elements of a set are of the same type.

Constructors

It is possible to declare new constructors and deconstruct values which are made of these constructors using the case-construct. An example using peano arithmetic is shown in Listing 5.4.

5.2.2 Full Language

The Full language extends the Core language with a lot of syntactic sugar. The process of converting the Full language to the Core language is called desugaring. We will take the graph coloring problem as a leading example throughout this section. The formulation of a graph coloring problem in FML can be seen in Listing 5.5.

In the Full language, we discern three different kinds of statements: declarations, definitions and constraints. Declarations introduce new symbols for which the interpretation is not given through a definition (e.g., symbol colorof). Definitions

Listing 5.5: Graph Coloring (in the Full language)

```
//Definitions of given sets
borders := {("a","b"), ("b","c"), ("c","a")}.
colors := {1..3}.

//Declaration of the interpretation we are looking for
colorof/1 :: function to colors.

//Constraint: For all borders (x,y) the color of x
//            should be different from that of y
! borders (\(x,y) -> colorof x ~= colorof y).
```

introduce new symbols of which the interpretation is fixed (e.g., the border relation). Constraints are boolean expressions which must be true in every model (e.g., adjacent nodes have different colors).

Declarations

Declarations have the form: `name :: declarationkind.`. The most important declarations are:

- `e :: element of set` where `set` is an expression evaluating to a set. The interpretation of `e` will be a single element of the interpretation of the set.
- `s :: subset of set` where `set` is an expression evaluating to a set. The interpretation of `s` will be a subset of the interpretation of the set.
- `f/arity :: function to set` where `set` is an expression evaluating to a set and `arity` a natural number. The interpretation of `f` will be a function of arity `arity` with codomain `set`.
- `c/arity :: constructor` where `arity` is a natural number. The interpretation of `c` will be a new constructor function of arity `arity`.

There are also extra declarations like `proposition` or `predicate`. These can be rewritten in terms of other declarations such as `element of {true,false}` or `function to {true,false}`

Example 5.2.1. A graph coloring problem needs one declaration: the coloring of the nodes, which is a function to colors. ▲

Definitions/Constraints

A definition has the form `identifier := expression.`. A constraint is just an expression by itself. Definitions can be recursive and mutually dependent (e.g., mutually dependent odd and even definitions). There is builtin support for first order logic-connectives, quantifiers, arithmetic and set-expressions. The quantifiers forall (`!`) and exists (`?`) are slightly different from the traditional notation in first order logic. They are higher order functions with two arguments: the set they are quantifying over and a boolean function explicitating the property which needs to hold.

Example 5.2.2. A graph coloring problem needs two defined relations: the borders and the colors. These can just be given as a set expression. There is one constraint, which quantifies over all borders. We can deconstruct the tuples in the border-relation in place using `\(x,y) ->` ..... The body of the forall quantifier just states that the colors x and y should be different. ▲

Set Expressions

Set expressions (sometimes known as set comprehensions) are a language feature based on the set-builder notation of mathematics. The set expressions in the Full language are much more general than those of the Core language. The syntax is heavily inspired by the notation in Haskell and Python. Set expressions can be replaced by a series of map and filter operations on sets but allow for a more readable syntax. Point-wise application of a function f over a set s looks like: `{f x || x <- s}`. Which can be read as: the set of evaluation of `f x` where `x` is in the set `s`. Or the subset of s for which the predicate p holds can be written as: `{x || x <- s, p x}`.

5.3 General Workflow

A lot of inspiration for the general workflow of FMS came from GHC [114]. FMS can be seen as a compiler to Answer Set Programming and most of the steps in the compiling process of GHC is relevant to FMS. An overview of the workflow can be seen in Figure 5.1. In this section we go over these steps to investigate them in more detail, from the file containing an FML specification to the models.

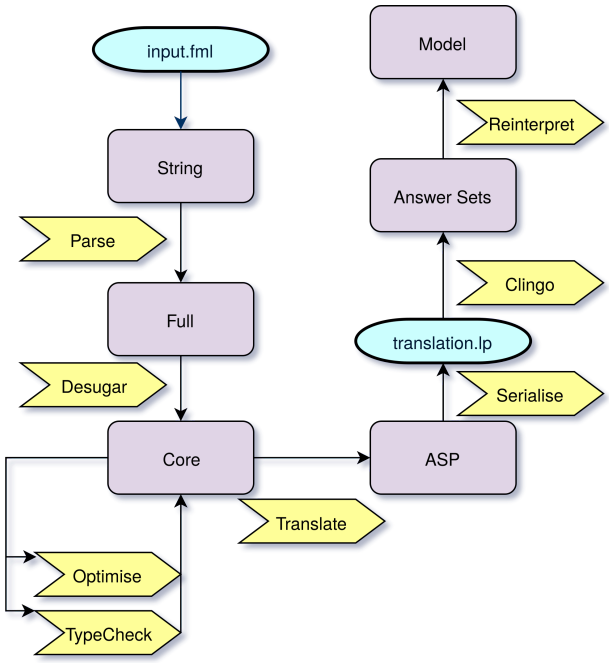

Figure 5.1: The general workflow of FMS

5.3.1 Parse and Desugar

Parsing an FML specification is done using the ANTLR4 [104] system. Based on a BNF grammar, ANTLR4 generates an Abstract Syntax Tree (AST) of an FML specification. This syntax tree is then desugared into the Core language, as defined in Section 5.2.1. Checking whether any identifiers are used out of scope is considered the last step in the desugaring process. Once a string is fully parsed and desugared, it becomes a self-contained well-formed Core expression without free variables.

Example 5.3.1. The expression `f 1 := 0. f x := 1`

is desugared to `f := (\y -> case y of 1 -> 0 ; x -> 1)` ▲

Example 5.3.2. The expression `(a,b) := f 5` is desugared to

```
temp := f 5.
a := case temp of (a,_) -> a.
b := case temp of (_,b) -> b.
```

The reasoning behind this desugaring is that all pattern matching needs to be done in the case-construct and definitions can internally only define a single variable, but the unsugared line defined `a` and `b` using a single assignment. ▲

Desugaring complex set expressions results in usages of the bind builtin. The bind operation is a standard operation for monads in Haskell [128]. In this case we consider the set monad. In this case, the bind operation is also sometimes known as flatmap. Flatmap (specialised for sets) boils down to a pointwise application followed by a flattening. Desugaring sets could potentially be further improved by using an applicative style instead of a monadic style wherever possible [90]. An applicative style omits the flattening part of the operation. Because of this, the operation requires less intermediate steps. This results in a smaller grounding and execution time. However, the applicative style has less expressivity so we need the general monadic style for at least some cases.

Example 5.3.3. The flattening of a set `ss` can be expressed by the following expression: `{x || s <- ss, x <- s}`

Which is desugared to: `bind ss (\s -> bind s (\x -> {x}))` ▲

Our approach directly transforms a FML specification to the Core language, which has some disadvantages. While the location information of symbols is stored inside the Core language data structures, some details about the original representation get lost. This means that every error which is detected in the file after desugaring is less clear than if the analysis would have been done on the original representation. This mainly concerns errors about out-of-scope identifiers and typing errors, these are exactly the ones which are checked on the desugared version in GHC [114].

5.3.2 Optimise

We could directly translate the Core language to ASP. However, there are many specification transformations which make the specification simpler. For this reason a number of optimisation techniques are applied to the expressions in the Core language. These optimisations deserve their own section and are further described in Section 5.4.

5.3.3 Type Check

Types have no semantic value in FML. This means that the evaluation of a specification can be done without taking typing information into account.

Listing 5.6: Translation of Listing 5.5

```
:-not bool(X),result(X).
member(s0,X0):-X0=1..3.
out("colors",s0).
{lamInter(l0,X2,X1):member(s0,X1)}==1:-lamDom(l0,X2).
out("colorof",l0).
member(s1,("a","b")).
member(s1,("b","c")).
member(s1,("c","a")).
out("borders",s1).
lamDom(l0,X4):-member(s1,X3),(X4,X5)=X3.
lamDom(l0,X5):-member(s1,X3),(X4,X5)=X3.
bool((b0,X3)):-X6<>X7,lamInter(l0,X4,X6),
    member(s1,X3),(X4,X5)=X3,lamInter(l0,X5,X7).
bool(b1):-not bool((b0,X3)),member(s1,X3).
bool(b2):-not bool(b1).
result(b2).
```

Nevertheless, types are an important tool to detect mistakes in a specification, both for the end-user who writes a specification, and for a compiler writer when implementing optimisations to check that the optimised version of the specification is still well-typed. A fairly standard Hindley-Milner [106] type system is implemented on the Core language (and not on the Full language). As explained in the last paragraph of Section 5.3.1, a second implementation on the Full language would improve the error messages produced by the type system. Note that doing type checks on the Full language does not fully eliminate the need for a type checker at the Core level, type checking on Core is still useful to detect errors in the optimisers. The error diagnosis in case of a badly typed expression is also very vague in the current state of the system. Many extensions of the standard algorithm exist which would lead to clearer error messages [129], a future version of the system would benefit from those.

5.3.4 Translate

The translation is arguably the most important part of the system. This part of the system does the job of translating the Core language to an Abstract Syntax Tree representation of ASP. Although the algorithm is complex, it is only a minor part of the actual code. The principles of this translation process were already explained in Chapter 4, but we still consider it interesting to show how the machine generated ASP actually looks. The ASP program we consider here

Listing 5.7: An Answer Set of the program in Listing 5.6

```
result(b2)                      lamInter(l0,"c",2)
bool((b0,("a","b")))            member(s0,1)
bool((b0,("b","c")))            member(s0,2)
bool((b0,("c","a")))            member(s0,3)
bool(b2)                        member(s1,("a","b"))
lamDom(l0,"a")                  member(s1,("b","c"))
lamDom(l0,"b")                  member(s1,("c","a"))
lamDom(l0,"c")                  out("borders",s1)
lamInter(l0,"a",1)              out("colorof",l0)
lamInter(l0,"b",3)              out("colors",s0)
```

is shown in Listing 5.6, which is the translation of the graph coloring example of Listing 5.5. The translation in the listing is slightly simplified for readability. The only difference lies in the omission of superfluous empty tuples

The ASP programs that FMS produces contain three kinds of statements. Declarations are translated to choice rules. Definitions and constraints are translated to rules and there is one constraint rule: it enforces that the evaluation of the constraints is true. This constraint can be seen on line 1. Sets are characterised by a name such as `s0`, and `member/2` represents the membership relation between sets and their elements. Line 2 in the translation defines the set `s0` as the numbers of 1 to 3. In line 3 you can see that this set represents the set of colors of the input. The `lamInter/3` relation is used to define the relation between a function (the first argument), an input (the second argument) and its output (the third argument). Line 4 defines `l0` as an uninterpreted function mapping every member of `lamDom` (the predicate which is used to represent the relevant domain of the function) to a member of the set `s0`. This corresponds to the colorof function, this is explicitated in line 5. Lines 6-9 do the same thing for borders as lines 2-3 did for colors. Lines 10 and 11 define that the relevant domain are the first and second components of the elements in the set of borders. Line 12-15 make up the translation of the constraint. Just like functions, booleans are given a name like `b0`. The `bool/1` relation is used to indicate whether the boolean is considered true. In line 16, it can be seen that the boolean `b2` is considered the top constraint.

As a whole this ASP program has models which correspond to the 6 possible colorings of the graph. One example answer set can be seen in Listing 5.7. Once the translation is obtained, we send this ASP program further into our workflow towards Clingo.

Listing 5.8: Outputted version of the answer set in Listing 5.7

```
{borders=[(a,b), (b,c), (c,a)],
 colorof=[(a,1), (b,3), (c,2)],
 colors=[1, 2, 3]
}
```

5.3.5 Clingo

We chose the ASP solver Clingo as a backend for our system. We communicate with Clingo via text files and use JSON output facilities of Clingo to be able to easily parse the answer sets. From the viewpoint of implementing FMS this is a very easy step. We print the ASP AST to a text file, call Clingo and reparse its output. The advantages of file-based communication are that it is easily debugged as it is human readable and it is easy to replace Clingo with another solver using the ASP Core [17] standard. It would be more efficient to directly use the internal APIs of Clingo to build the ASP program. Clingo produces answer sets, which represent models for the problem, but it is still not the end of our workflow.

5.3.6 Reinterpreting the Answer Sets

Answer sets map one-to-one to FML models. However, it is not obvious how the answer sets from Clingo relate to the original problem statement, as can be seen in Listing 5.7. The answer set possibly contains higher order functions and sets, which are relatively complicated when encoded in an answer set. The reason for this is that every set gets its own ASP identifier and the sets are fully determined through its definition of the membership relation. A set $\{\{2\},\{3\}\}$ could be represented as the following ASP facts: `member(s,s1), member(s,s2), member(s1,2), member(s2,3)`. The reinterpreation step reorders this information of the answer set into a more natural format.

Example 5.3.4. The answer set in Listing 5.7 is transformed to make it more readable for the end user. This result can be seen in Listing 5.8. ▲

In this step we also notice the consequences of file-based communications with Clingo. In most situations, not all information in the answer set must be presented to the end user. The current approach only lets us filter the information after the solver has finished and printed the complete answer set. A tighter integration with Clingo could overcome this problem.

5.4 Optimisations

FMS contains a number of optimisations which are transformations of Core expressions. Optimisations have the goal to make the expression that needs to be translated simpler. In most cases this has a direct correspondence to smaller expressions. Shorter expressions will lead to a smaller translation, which leads to a smaller, but equivalent SAT formula. This potentially results in exponential speedups. In this section we give an overview of the implemented techniques.

5.4.1 Stratifying Definitions

Let-bindings are potentially very large, containing definitions for a lot of different symbols. Translating mutually recursive definitions leads to extra intermediate symbols in the translation process. Smaller, but deeper nested bindings explicitate that there is no recursion between symbols and result in better translations. We can use a topological ordering of the bindings to transform the bindings. Tarjan's algorithm is used to find this ordering. This has already been done in other systems such as IDP3 [76] or Scheme compilers [127].

As a side effect of this topological ordering, definitions which are unused can be thrown away, because the formula after the "`in`" provides the starting points for the stratification. If a definition is unconnected to the formula, it is not included in the topological ordering.

Example 5.4.1. `let odd x := even (x-1) ; even x := if x = 0 then true else odd (x-1) ; c := 4 ; e := even c ; d := 8 in e`
can be stratified into:
`let odd x :=... ; even x :=... in (let c := 4 in (let e := even c in e)).` ▲

Benefits Stratification potentially removes unneeded definitions. These definitions do not need to be translated and thus are prevented of taking up time or space in the actual translation or solving process.

5.4.2 Inlining

Let-bindings contain definitions for certain symbols. These symbols occur in the inner expression of the let-binding. Sometimes it is beneficial to replace the defined symbol with its definition. It allows for other optimisations and sometimes even lets us eliminate the definition completely. The process of replacing a symbol with its definition is called inlining. Inlining is done in a lot

of major compilers. The GHC compiler for Haskell excels in this [78] and we copied some of their techniques into FMS. One simple observation is that if a binding only has one single usage, it is always better to optimise the binding for this usage, as we can remove the original binding because it becomes unused.

Example 5.4.2. In the expression `let y := 2*x in y+5`, the variable `y` can easily be inlined, so the new expression becomes: `2*x+5`. ▲

Example 5.4.3. In the expression `let y := f x in y+y`, the variable `y` would not be inlined, as `f x` would occur twice in the resulting expression, which could duplicate work in further steps. ▲

Another inlining rule is that bindings to constant integers or strings should always be inlined, as handling those is generally not any more complex than handling a variable, but allows for more specialised optimisations. Sometimes the end user knows that inlining a function would be beneficial but the compiler could not derive this. For this reason FML allows for compiler directives which force the compiler to inline (or not inline) certain functions.

Example 5.4.4. The directive in `let f # inline; f x := true in f 4.` forces f to be inlined, so this expression can be fully reduced to `true`. The directive in `let f # noinline; f x := true in f 4.` prevents f from being inlined, so this expression will be translated to ASP in its original form.

▲

Benefits A let-binding can make it harder to see opportunities for optimisation such as boolean simplifications of the next section. Inlining `f` in `let f := not in not (f p)` is essential to be able to perform the boolean simplification optimisation.

5.4.3 Boolean simplifications

When handling complex nested boolean formulas, applying some boolean rewrite rules can simplify them greatly. For this reason, pushing of negations is implemented just like in IDP3 [36]. This technique uses the standard rewriting rules for propositional and first order logic.

Example 5.4.5. The boolean expression `not (or (not p) (not q))` is simplified to the expression `and p q` ▲

Benefits Expressions with more nested function applications will lead to more calculation steps for the evaluation of the expressions. The expression `not (not q)` will always have the same value as `q`. However, after translation the ASP solver would need two propagation steps to obtain the truth value of the former and immediately know the truth value of the latter. Note that it is possible that the value of `q` can potentially change a lot of times due to the backtracking in the SAT solver component of the solver. So the netto effect of this optimisation is much bigger than two propagation steps.

5.4.4 Constant folding and beta reductions

Constant folding [4] is a common optimisation technique for compilers. Whenever the arguments of builtin functions like addition or multiplication are fully known, we can replace the expression with its evaluation. This can be extended to more complex expressions: if the "if"-part of an if-then-else expression is trivially true, the if-then-else can be replaced by just the then-part.

When an anonymous function is applied to certain arguments, we can specialise the function body with the given arguments. This corresponds to the concept of beta reduction in the lambda calculus. Using beta reduction prevents the overhead of translating an extra lambda and can pair up with other optimisations such as constant folding for more efficiency gains.

Example 5.4.6. In the expression `(\x -> x + 4) ((\x -> 5) a)`, the second part `((\x -> 5) a)` can be beta reduced to `5`. The resulting expression `(\x -> x + 4) 5` can be beta reduced to `4 + 5` which can be reduced to `9` through constant folding.

▲

Benefits The benefits of this technique are the same as for the boolean simplification case. This is a generalisation which works for non-boolean expressions and leads to the same prevention of propagation steps.

5.4.5 Combining the optimisations

All of the above optimisations can rewrite part of the expression tree. The application of an optimisation possibly opens up possibilities for other optimisations or sometimes, even the same optimisation. To fully optimise an expression, all optimisations should be repeated until a fixpoint on the expression is reached.

However, it is possible that some interactions between optimisations could lead to an infinite chain of optimisations. So for practical purposes we limit the number of iterations of the optimisation transformations.

5.4.6 Future Optimisations

All above optimisations are already present in the current system. There are also a few optimisation techniques which have not been implemented but could be interesting future expansions.

Rewrite Rules

Boolean simplifications are applications of rewrite rules which make the expressions simpler. If the user writes new complex functions, it can be beneficial that the end-user is able to write such rewrite rules for his own functions as meta-information for his definitions. This technique would be able to generalise the boolean simplifications and make it applicable to more situations. Such rules are already available in some programming languages such as Haskell.

Example 5.4.7. A rewrite rule like `not (not x) => x` could explain that double negations can be ignored. ▲

Ground with Bounds

There are a lot of optimisations for grounding researched in the context of the IDP3 system. While some are subsumed by new techniques which are explained in this section, a lot of them are also applicable for the FML system. Grounding with bounds [131, 132] is such a technique that proved to be signficant for the performance of IDP3. Grounding with bounds uses symbolic reasoning to limit the size of quantifications. It is our hypothesis that these techniques generalize naturally to the higher order case of FML. This would allow quantifications over implicit domains so the forall quantifier could have type `(a -> Bool) -> Bool` instead of type `Set a -> (a -> Bool) -> Bool`.

The current purpose of this first argument of type `Set a` is to limit the number of instantiations we have to make for the quantifier. The ground with bounds algorithm could derive a small enough domain, so that this extra argument is not needed anymore.

Example 5.4.8. Ground with bounds could rewrite an unbounded quantification like `! (\x -> (5 < x & x > 10) => p x)` to a bounded expression `! {6..9} p` which can be handled by the algorithms which are already in place. ▲

5.5 Dependencies and availability

The system is fully written in Kotlin, with a Gradle build system, in conjunction with a set of libraries. The most important ones are ANTLR4 for parsing, logback and slf4j for logging, JUnit for testing, and klaxon for JSON communication.

The software and its source can be found online at `https://dtai.cs.kuleuven.be/krr/fms`. There is support for testing the software online on the website itself or you can download it yourself. The only external dependency is Clingo, which can be downloaded from `https://potassco.org/clingo/`.

At the moment there is no IDE support for FML, but the development of an implementation for the Language Server Protocol[91] is in the early stages of development. This is a generic framework which editors can support to automate the integration with a new language.

5.6 Conclusion

This chapter introduces FMS and its general internal workflow. As far as the authors are aware this is the first attempt for using a higher-order functional language as a modelling language. Section 5.2 introduced the Core language and its syntactical extensions into the Full language. We touched upon the different steps in the solving process and on the optimisation techniques which are available in the system. We also highlighted points where further work could significantly improve the system. A thorough comparison with other systems such as SMT or ASP is still lacking.

Part III

Tooling for IDP3

About This Part

Tooling is an important factor in the usability of any (programming) language. Good tooling can lower the barrier of entry for new users and help expert users to be more productive. Commonly available tooling include source code editors, debugging facilities, formatters, code linters, ... For many languages, there are also Integrated Development Environments (IDE) available. These IDEs bundle a set of tools into one easy-to-use program.

Apart from tools for writing new software, enabling an easy integration with other systems can be useful. In most cases this takes the form of a library, e.g., most programming languages have bindings to one or more GUI libraries.

This third and last part of the thesis contains two chapters and each one focusses on one tool that was developed for the IDP3 system. Chapter 6 is about the web-based IDE that was developed for IDP3. Apart from the standard advantages of having an IDE, this IDE is web-based so an online version is made available. This enables potentials users of the system to try the system online without installing any software.

Chapter 7 is an interactive configuration tool that allows to interactively explore the solution space of an IDP specification. This tool also shows how logics with a model semantics can be used to do more than specifying model expansion problems. It was originally developed to show these ideas in the context of Decision Model and Notation, but it is applicable to all specifications made in IDP.

6 A web-based IDE for IDP

A first version of the work of this chapter was presented at the International Workshop on User-Oriented Logic Programming 2015 [31].

6.1 Introduction

IDP[37, 16] is a knowledge base system based on an extension of first order logic: FO(·) [43]. The goal is to split classical programs up into two parts: a declarative part (the domain knowledge) and an imperative part (the tasks done with this knowledge).

The language has proven promising in practical situations, but to better support new and existing users, an IDE is an indispensable tool for any new language. In order to facilitate this need, we developed a web-based IDE for FO(·). The tool is available online at `http://dtai.cs.kuleuven.be/krr/idp-ide/`. This new IDE supports the basic features you would expect from an IDE such as syntax highlighting, error/warning visualization, and auto-indentation. On top of this there are also a few features specific to IDP. An example is the unsat core visualization, this helps the user debug a program by indicating what part of a theory is inconsistent. The editor has been used successfully in a course teaching IDP at the KU Leuven, and the online version has already attracted new users for IDP who use it in e.g., the tax administration business.

IDP had one previous IDE[74]. It was based on the Eclipse Environment [48] and XText [133], however it was tedious to keep this IDE up to date with new Eclipse versions and IDP language updates. The new IDE uses a more lightweight approach, while relying on the maturisation of web technologies. This enabled us to provide more features with a smaller code base and to host

an online version of the IDE, which can be integrated into a website to present the system.

In the following sections we give an introduction to the IDP system, an overview of the features of the IDE, give some insights in the implementation of the IDE and finally compare the IDE with other IDEs for systems that are similar to IDP.

6.2 The IDP system

IDP programs typically consist of 4 objects. A vocabulary, theory and structure are used to declaratively define the knowledge which is relevant for the program. The procedure then further explains what the system should do with this knowledge.

Vocabulary The vocabulary declares which types/predicates/functions will be used.

Theory The theory is the representation of the knowledge over a vocabulary, expressed in FO($\cdot$). It makes use of the standard first order concepts such as $\forall, \exists, \Rightarrow, \wedge, \vee, \neg$ extended with arithmetic, types, and inductive definitions.

Structure A structure is a (partial) interpretation of the vocabulary. It should at least interpret the types of the vocabulary and can further interpret as little or as much of the rest of the vocabulary as desired. Models of a theory fully interpret all symbols so that the theory is satisfied.

Procedure IDP uses a Lua scripting environment to interact with the above objects. A typical procedure would be to print the models of the theory, but this procedure could also include an interactive simulation of a system which is described in the theory.

IDP offers a lot of built-in procedures - inferences - which operate on the above objects. Some examples of these are:

Modelexpand(Theory, Structure) This inference produces a model satisfying a theory and expanding a partial structure. The most typical main procedure consists of printing the models returned by this method.

Unsatcore(Theory, Structure) This inference produces an unsat core, which is a subset of the theory which is inconsistent with the structure. This is useful for debugging purposes.

Propagate(Theory, Structure) This inference produces a more refined structure which approximates every model of the theory which expands the original structure.

Progress(Theory, Structure) Returns the set of possible next states (structures) for a linear time calculus (time dependent) theory. This can be used to simulate a dynamic system as explained in [13].

In general, most of the time working in IDP is spent while writing theories. Procedures are traditionally quite short and mainly consist of calling one of the inference procedures.

6.2.1 Approach to the IDE

The IDP system is comparable to ASP systems, for which a multitude of IDEs were developed, which are explored further in Section 6.4. Our approach differed from most of the others in the sense that we tried to develop a more lightweight system. A lot of the IDEs try to develop a lot of tools from scratch. However, good frameworks are already available for developing IDE tools. We chose for a web-based approach, based on existing frameworks. This approach provided us with some advantages.

A web-based tool is agnostic about the operating system it runs on. This means that no special code is needed differentiating the different operating systems, while still providing a modern-looking layout.

Another advantage is that we can provide the IDE as a local application as well as a website. The local application is based on a locally run web server. The workspace folder and the IDP run command can be set via a configuration file. The application can be accessed by browsing to localhost. The web application is the same tool as the local application with some added restrictions. The workspace folder is not writable via the application and the resources for running files are limited.

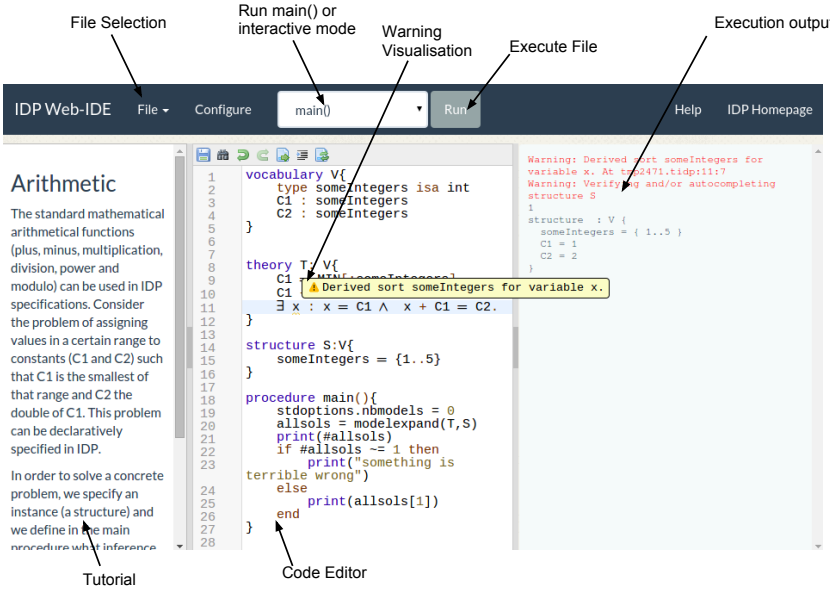

Figure 6.1: User Interface of the IDE

6.3 Overview of the IDE

6.3.1 Features

Editing

Most of the time a user spends in an IDE, he is editing the source code. It is the responsibility of the IDE to support the user as well as possible. Apart from the obvious editing of text, a user expects facilities such as highlighting errors and auto indentation.

Basic Text Editing The IDE provides all the features you would expect from a simple text editor. This includes picking a file to edit, providing a text area to edit the code, and supporting the most common general editing features such as undo/redo, find/replace and line numbers.

Autocompletion/Code Snippets IDEs have often very different notions of autocompletion. Some IDEs perform a full semantic analysis to suggest the possible correct completions of the current word. The autocompletion in this IDE is more simplistic. It is based on all the words occurring in the current file combined with known code snippets. These snippets are predefined and consist of a set of common pieces of code. These include declaration of components such as theory and vocabulary, and for example a reachability relation. Autocompletion can be triggered through the Ctrl+Space shortcut.

Syntax Highlighting Syntax highlighting is done on a syntactical basis. The theory is divided into tokens, they indicate the function of the word in the source coding, such as logical symbol, comment or keyword. This information is then used to color certain parts of the code.

Logical Symbol Replacement IDP makes heavy use of ASCII representations of mathematical symbols. The tokenized version of the theory identifies the logical symbols. The editor automatically replaces these symbols with their mathematical counterparts.

Auto indentation Uniform indentation vastly increases the readability of files. The editor supports the user to use a consistent indentation and provides the possibility to reindent the whole document.

Error/Warning visualization Syntax errors and warnings are visualized right where they occur through zig-zag lines. Without this feature, users spend a lot of time tracking down the exact location of an error. This feature is implemented by running the internal parser of IDP, so the feature always uses the current version of the syntax. This prevents the IDE from being unusable when a new language construct is added to the language.

Code Sharing The editor (both local and server) provides the capability to export code and provide a global link for sharing the code. This part of the editor uses a GitHub Gist-account[63] to back-up the code.

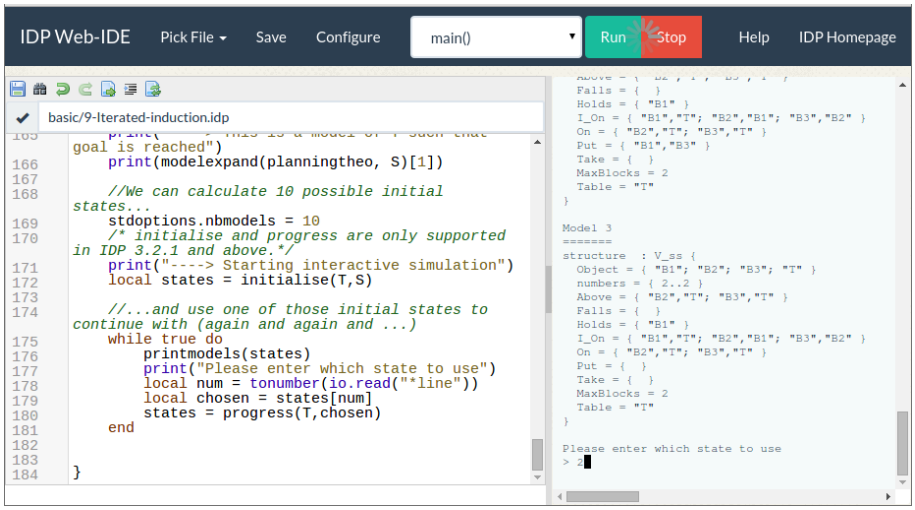

Figure 6.2: Interactive Running

Running

Apart from editing, the IDE can also provide enhanced support for running the files a user wrote. The most basic support is simple terminal support. But this IDE also supports some extra visualization features.

File Running Executing a file is supported in the IDE. This includes the possibility of an interactive session, or a program where a Lua procedure needs to ask input from the user as can be seen in Figure 6.2.

Lua Shell Mode Apart from running the main procedure of a specific file, it is also possible to select "Interactive Mode" in the dropdown menu to the left of the run button. This starts an interactive Lua Shell so the user can interact directly with the components in the file.

Unsat core visualization IDP supports the user throughout the debugging process of unsatisfiable theories with the help of unsat core extraction [112]. This process extracts a subset-minimal part of the theory which is inconsistent over a given structure. The result is thus a set of ground formulas which are instantiations of lines in the theory. This information can be visualized in the editor with zig-zag lines at the appropriate theory lines together with a tooltip

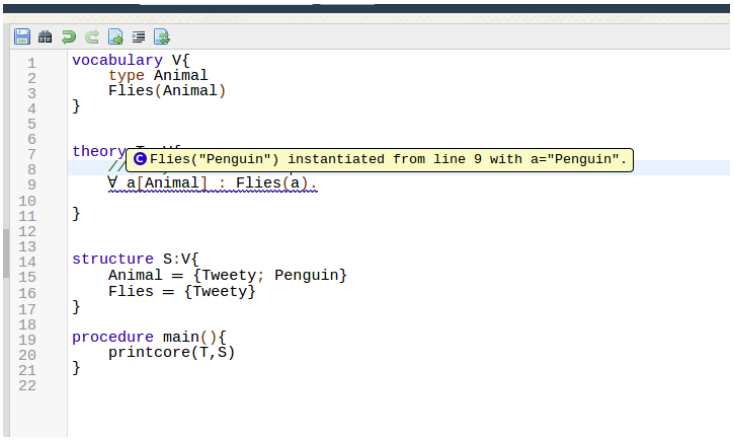

Figure 6.3: Unsatcore Visualization

indicating the instantiations in the unsat core. This visualization can be seen in Figure 6.3. In this example, the structure says there are 2 animals, of which one can fly. This is not consistent with the theory stating that all animals fly. The core explains the unsatisfiability of the theory with the instantiantion of the sentence where the unsatisfying animal is the penguin.

Visualizations IDP provides the possibility to write theories over a graphical vocabulary so that the output can be visualized. This visualization can even be interactive, so it can react to clicks of the user. This visualization technology is called IDPD3 [84], it uses the same philosophy as ASPVIZ [23] and IDPDRAW [73] extended with interaction. This technology is mostly used to visualize the output of a search problem but can also be used to provide an interactive puzzle to a user, as can be seen in Figure 6.4. When a visualization method is called, a pane with the image is automatically opened.

Online-Specific Features

A few modifications are used for the online version of the IDE. There is one extra feature available: the availability of a tutorial mode. In the online version of the IDE, a pane with explanations can be automatically coupled to an example. This way, a tutorial with examples can be completely integrated with the IDE, so the IDE can be used as a learning environment. The public website of the IDE is structured in this way, an example of this can be seen in Figure 6.5.

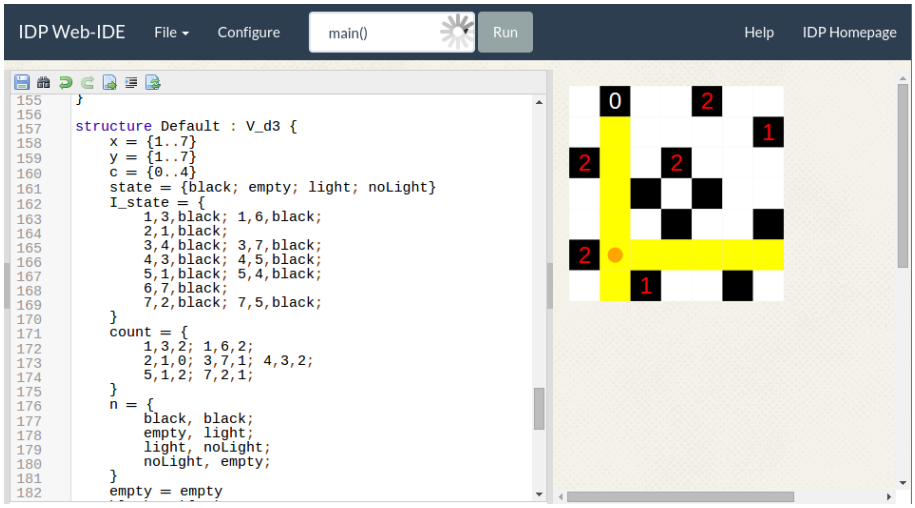

Figure 6.4: Visualization of an interactive Lights Out puzzle

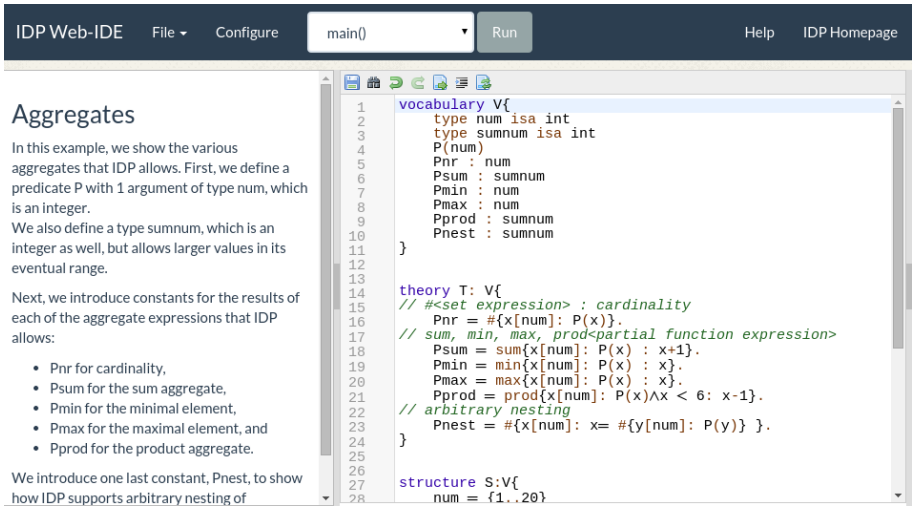

Figure 6.5: Tutorial mode of the editor

The security aspect rather enforces a restriction of features. When providing a system online, precautions need to be taken so the system is safe. In a standard setting the Lua environment of IDP provides full access to the host system, so unsafe methods should be disabled. Also resource limitations are imposed upon requests coming from the web.

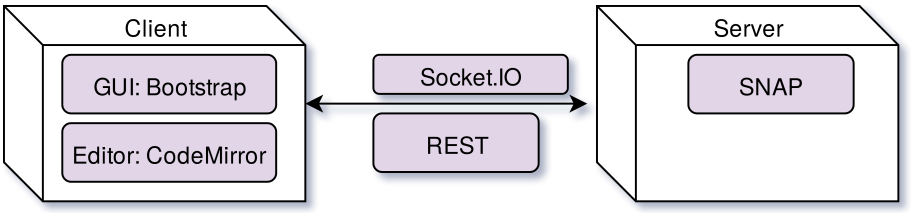

Figure 6.6: Overview of the IDE architecture

6.3.2 Implementation

The client side of the IDE completely consists of JavaScript, CSS and HTML code and the server is written in Haskell. Libraries were used for the generic components of the UI. The general structure of the IDE can be seen in Figure 6.6.

CodeMirror CodeMirror[25] is a JavaScript-based text editor with an API which has good support for extensions and new languages. All text-editing features are implemented on top of this editor. Features such as syntax highlighting are natively supported and only require simple, language-specific code to be added.

Bootstrap This state-of-the-art CSS/JavaScript framework developed by Twitter [14] allows for the quick creation of a simple and effective user interface. It is well supported by all browsers, so we can ensure that the IDE works equally well on all operating systems.

Snap To communicate between the CodeMirror instance in the web browser and IDP, we wrapped IDP into a RESTful interface. This is the most common technique on the web for exposing an API. This interface is written in Haskell using the Snap[115] framework.

Socket.io When running an interactive IDP session, we needed to keep a session open during its runtime. To support this communication, we chose for WebSockets. This is a fairly new technology, but luckily socket.IO[116] provides an automatic fallback mechanism whenever a browser does not support WebSockets. In this case, the application will run through traditional XHR/JSONP requests which simulate the WebSockets.

With the use of these technologies, the actual code of the IDE can be very compact, e.g., the web server consists of less than 300 lines. We also expect that future changes to the IDP language require very little change to the IDE

as features such as the error reporting are fully dependent on the parser of the IDP executable. This approach to parsing also has disadvantages. It is difficult to implement advanced features such as smarter autocompletion without having a full parser available in the IDE itself. We judged that for this editor the costs did not outweigh the benefits.

6.3.3 Extensions through Semantical Analysis

As indicated in the previous section, the IDE is kept very lightweight. This makes it more resistant to changes and keeps the development time low. However, this means that the IDE can only support the user on a very high level. There are a number of features which are available in advanced editors which we an can only provide through an IDE which includes semantic support:

Refactorings Common refactorings include the renaming of a particular symbol. A full find-and-replace on the source can approximate this action, but this does not work if the same name occurs in for instance a string constant. The editor needs to have semantic knowledge about the symbol to properly support the kind of refactorings.

Code Completion It is useful that the IDE suggests which symbols you can use in a particular place. The IDE needs to be aware of the types of the symbols and the type of the current expression that the user is forming.

Find References If a symbol is declared, it can be useful to quickly see in which places this symbol is actually referred to. An advanced IDE could even warn when there are declared symbols that are completely unused.

6.4 Other IDEs

Existing editors for truly-declarative languages such as FO(·) and ASP can be roughly divided in two groups: custom IDEs which are built from the ground up and Eclipse plugins. This IDE falls into a third category, the web-based IDEs.

6.4.1 Custom IDEs

Custom IDEs often lack some basic features such as find/replace or undo/redo, which vastly undermines their practical usability. An example of such an editor

is OntoDLV [109], which does not support undo. ASPIDE [53] seems to be the most comprehensive custom IDE which does include basic features such as find and undo. It has an impressive feature list, including quick fixes for code. However, the use of Java GUI tools causes the GUI to be platform independent. This is easy from the programmer point of view, but OS-specific things that users expect in all applications, such as shortcuts, are not there.

6.4.2 Plugins

Some editor can be extended to support new languages using plugins. This can significantly reduce the development time as many generic functionality is usually available in these editors and the plugins only need to provide the language specific functionality.

Eclipse Eclipse plugins benefit from the good integration Eclipse provides for new languages. The previous IDP-IDE was developed as an Eclipse plugin using Xtext. The disadvantage of this approach is that Xtext requires a complete new parser for the language. This means that the IDE needs an update at every syntax change of the language. It is sometimes difficult to ensure the real language and the Eclipse language consist of the same grammar, thus it occurs often that either the parser is not strict enough and some syntax errors go unnoticed until the file is run, or some errors are indicated which are actually acceptable constructs.

Examples of ASP editors in this category are Videas [101] and SeaLion [102]. This IDE has its own visualisation tool Kara [83]. SeaLion seems like the most modern ASP IDE. It has a tight integration with Eclipse, and provides features such as a debugger and visualizations.

Language Server Protocol Microsoft presented in 2017 an even more generic approach to adding languages to editors using the Language Server Protocol[91]. Their aim is to have a common interface between all editors and all languages so that if an editor supports the Language Server Protocol, it supports all languages which support the same protocol.

This approach has seen some adoption, having support for popular editors such as Sublime Text 3 and IntelliJ and most of the popular programming languages. However, there is no implementation for modelling languages such as ASP or MiniZinc at the time of writing. If this standard was available at the start of the development of the IDP web-IDE, this approach would probably have been taken.

6.4.3 Web-based IDEs

Web-based IDEs are a relatively recent phenomenon, but have gained some traction in the last few years. Cloud9[24] is one of the more popular ones and supports most of the mainstream programming languages. It provides the user with all tools he uses during development of an application, including full access to a virtual machine. The IDP web-IDE belongs to this category.

Our IDE has one advantage that most of the others have not. It is one of the few tools which support both offline and online use. Two other (general purpose) IDEs that support this are Codiad [26] and ICEcoder [72]. Both tools use PHP for their back-end and ace-editor for front-end editing. Their editing features are more extensive than our IDE, especially the support for folders and projects. However, there is no integration for running the files which are being edited.

Some ASP systems do have an online tool available. The DLV system has an online version since 2006 available at `http://asptut.gibbi.com/`. This website mainly consist of a plain textbox and a run button. It mainly serves the purpose to follow a tutorial in the form of slides. Clingo has a similar tool available at `http://potassco.sourceforge.net/clingo.html`.

In more recent years the ASP community has started working on more comprehensive web-based IDEs. Such as LoIDE [61] or an IDE around SPARC[88], a language which extends ASP with sorts. These editors are partly inspired by the web-IDE for IDP and contain roughly the same feature set for the different languages.

One system in this category is special. Atom [6] is a purely offline editor which is also based on web technology. It essentially consists of a web browser with less security restrictions, so it integrates better with the local filesystem, but there is no way to run Atom as a web service. Atom is not inherently coupled with any language or system, but supports many plugins for a multitude of languages.

6.5 Integration in other web applications

An advantage of our approach is that it makes it possible to include parts of the IDE into other web-based applications. Inserting an IDE in an application is as simple as loading the CodeMirror library with the right set of options. Running a file can be done through the REST interface which is exposed by an IDE server.

We currently have multiple demos online which make use of this approach. An overview can be found at `http://krr.bitbucket.io/`. The most important ones are the Europe Coloring and the Course Selection demo. The first one demonstrates a map coloring problem, integrated with the map of Europe. This demo is available at `https://krr.bitbucket.io/eurocol/`.

The second one demonstrates the selection of courses in a study programme. Through the inclusion of the editor we can provide the possibility for the user to interactively modify the underlying theory of this course selection system. This demo is available at `http://krr.bitbucket.io/courses/`.

6.6 Conclusion

In this chapter we presented the IDP Web-IDE. This new IDE for the IDP system was developed using web technology. This enabled us to provide an online as well as an offline version of the editor. The IDE supports most of the basic features you would expect from an IDE. We believe that the web-based approach is a new setting for IDEs and that this setting works inspiring e.g., to add new features. The recent addition of the tutorial mode is such an example.

However, some features from other IDEs, such as managing multiple files as a project are currently not supported. We intend to expand the the IDE so it can continue to serve as a teaching and development tool, but the approach will remain focused on a low effort tool. We also want to make the tools in the IDE available for other users so it becomes easier for other developers to make tools like in Section 6.5.

The IDE is publicly available at `http://dtai.cs.kuleuven.be/krr/idp-ide/`. There, the online version can be tried and a link to download the IDE for offline use is provided.

7

Combining DMN and the Knowledge Base Paradigm for Flexible Decision Enactment

A first version of the work of this chapter was submitted for the RuleML Challenge at the International Web Rule Symposium 2016 where it was presented and won the Challenge Award. The contents of this chapter are published in the Supplementary Proceedings of the same conference [33].

7.1 Introduction

Representing business rules and the rules governing businesses is a challenging task [126]. Supporting the enactment of the represented rules poses even greater challenges. Many representation forms for both business rules and decision management have been proposed [64, 65, 67, 103], as well as several rule engines aimed at the execution of business rules [57, 96].

We present a novel approach for representing and enacting business rules and decisions. Our approach leverages the simplicity and expressive power of the Decision Model and Notation (DMN) standard to represent decisions and the capabilities of a knowledge base system, equipped with multiple inferences [37], to enact them. A user-friendly graphical user interface allows for interactive decision enactment on the modelled decisions.

The well-known uServ case study [54] is used to illustrate our approach. The rules entailed in the case study were represented in the DMN standard. To enact the represented decisions our approach utilizes a knowledge base system, IDP. In IDP knowledge is represented using an extension of first-order logic, FO($\cdot$) language. By utilizing the available inferences of the IDP system we show how the capabilities of current decision management solutions can be

expanded. In particular we show how the IDP knowledge base system can be used as an inference engine for DMN. Additionally we discuss several extensions to the uServ use case which, to the authors' knowledge, are difficult to achieve in existing solutions. The system can be accessed on `http://krr.bitbucket.io/autoconfig/`.

Our proposed system guides the user through the decision enactment process by automatically propagating the user's input, and thus immediately showing the consequences of their choices. In many rule languages, rules are computational devices computing heads from bodies; information flow is from bodies to heads. In the FO(·) language underlying the IDP system, sets of rules are definitions. Rules *define* heads in terms of bodies, but do not impose a computational direction. The IDP system can compute the defined concepts from known input, but it can also compute possible values for unknown input, given a known value for the defined symbols. As such, the information flow can go from body to head or from head to body, or a combination of both.

The remainder of the chapter is structured as follows. Section 7.2 provides a background on the recent DMN standard, and on knowledge representation and in particular the IDP knowledge base system. The used example, the uServ case, is detailed and representations in both DMN and FO(·) are provided in Section 7.3. Section 7.4 gives an explanation of the approach. Several scenarios for enactment are provided in detail in Section 7.5. An overview of possible extensions to our approach is given in Section 7.6. The chapter is concluded in Section 7.7.

7.2 Background

The approach proposed in this chapter uses a knowledge base to represent the decision rules usually maintained in decision tables or other decision or business rules management solutions. While knowledge representation and decision management emerged in different fields, they share similar goals. Both domains are concerned with finding good representations for knowledge. In the case of decision management this information entails the business decisions of an organization. In the domain of knowledge representation the information is represented in a general manner and can be used in several ways, e.g., to enact decisions. In both domains the emphasis is on ensuring the represented knowledge is actionable.

This section provides some background of both decision management and knowledge representation. The next subsection describes the newly proposed decision management standard, DMN. Subsection 7.2.2 describes the field of

knowledge representation, and in particular the knowledge base paradigm used in our approach.

7.2.1 Decision Model and Notation

DMN, Decision Model and Notation, is a new standard managed by OMG, the Object Management Group, to describe decision logic in a vendor-independent language and executable notation [103, 121]. Without DMN, the logic of decisions (pricing, eligibility, loans, compliance, ...) is all too often described in imperfect requirements documents and then translated into enterprise systems, with numerous disadvantages, such as: traceability problems, long revision cycles, interpretation errors, poor consistency checking, lack of flexibility, etc.

DMN provides a tool-independent decision modelling notation, and is giving rise to a class of executable modelling tools with the ability to specify and execute business-friendly and verifiable decision models. Key features of DMN are: (1) Decision Requirements Diagrams (DRD) describing the dependencies of a decision on other supporting decisions and information sources and (2) the decision logic, typically either a decision table or an expression in a new expressions language (FEEL), usable by domain experts but rich enough to handle real-world decision logic. Decision tables have been around in various forms for decades, and DMN imposes a number of constraints on their format.

7.2.2 Knowledge Representation

The field of knowledge representation and reasoning is aimed at developing fundamental scientific understanding of knowledge and its uses to solve problems. Thanks to spectacular progress in solver technology in the fields of SAT [10], constraint programming [5], Answer Set Programming [7] this field is reaching maturity. This can be seen in the development of increasingly expressive formal specification languages (overlapping or extending classical logic, inductive definability, higher order aggregates and quantification, bounded arithmetic, etc.), an improved understanding of expressing expert knowledge as formal specifications in these rich languages, an increased understanding of how computational tasks can be decomposed in different inference tasks and the development of a range of effective inference engines for solving various types of tasks from formal declarative specifications.

The area is broadening its view to new areas of application. Business Systems present exciting challenges to the field of KRR: realistic applications of knowledge based nature, of great practical and economic interest and posing

many challenging research questions because of the great variety of human knowledge and the need for modularizing it [62], the need for a range of different functionalities [100] (to compute decisions and run business processes), for verification tools (for compliance checking and others) [27, 62, 130] and the need of efficient scalable reasoning.

Most languages in these fields are designed with focus on a single or a few forms of inference. The aim of the FO(·)-KB project [37] is to build a logic (FO(·)) extending classical logic (FO) with language constructs of proven value from various KR languages and to develop the knowledge base system IDP to manage an FO(·) knowledge base and use it to solve a range of problems using various forms of inference. The value of being able to reuse the same declarative knowledge base for a range of functionalities was demonstrated recently in [70], where 8 different forms of inference reuse the same knowledge base to provide a flexible user-friendly interactive configuration system. More information on the IDP system can be found at `https://dtai.cs.kuleuven.be/software/idp`.

7.3 The Running Example

7.3.1 DMN

The dependencies, i.e., information requirements of the decisions in the uServ case, are represented using a decision requirement diagram (DRD) in Figure 7.1. Square boxes, such as the "Eligible for Insurance" box, denote decisions, while rounded boxes, such as "Price", denote input elements. The arrows show the direction of the information requirements, e.g., the Eligible for Insurance decision requires the result of the Client Score decision and information on whether or not the client is a long term client.

For reasons of clarity the DRD shown in Figure 7.1 is limited to the case of insurance policies covering a single car and driver. Adding support for multiple cars and drivers requires the use of business knowledge models, which represent reusable pieces of knowledge. For this particular example this would significantly increase the amount of elements in the DRD.

The description of the logic of each decision is then detailed at the decision logic level, using decision tables, boxed expressions, and the FEEL language.

In Figure 7.2 the decision table of the Potential Theft Rating decision is shown. As is evident from the DRD this decision requires three inputs: whether the car model is on the High Theft Probability Auto list, whether it is a convertible,

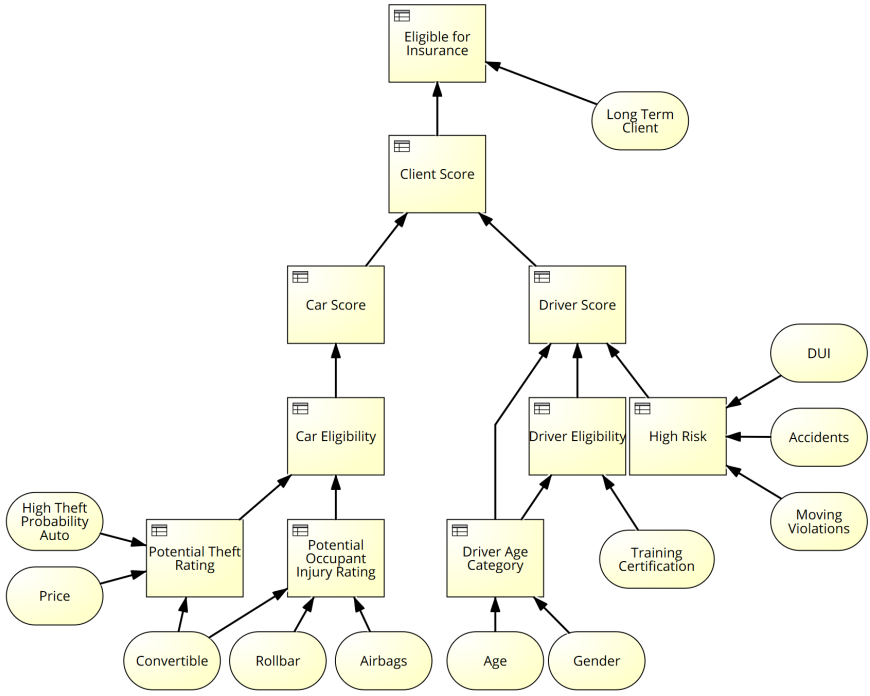

Figure 7.1: Decision Requirement Diagram for the uServ case, for the case with a single car and single driver.

and the car's price. These elements are represented by the columns of the table. Each row then corresponds to one of the rules detailed in the uServ case.

U	High Theft Probability Auto		Convertible		Price		Potential Theft Rating
	Boolean		*Boolean*		*Currency ($)*		*{high,moderate,low}*
1	=	true	-		-		high
2	=	false	=	true	-		high
3	=	false	=	false	≥	$ 45000	high
4	=	false	=	false	∈	$ [20000..45000]	moderate
5	=	false	=	false	<	$ 20000	low

Figure 7.2: Decision table containing the decision rules for the Potential Theft Rating decision.

7.3.2 FO model

The knowledge represented by the DMN model described in the previous subsection can be translated to FO($\cdot$), in the form of an inductive definition as follows.

$$\left\{\begin{array}{ll}
\forall c\,[Car] : PTR(c) = High & \leftarrow HighTheftProbList(CarModel(c)) \\
\forall c\,[Car] : PTR(c) = High & \leftarrow Properties(c, Convertible) \\
\forall c\,[Car] : PTR(c) = High & \leftarrow Price(c) > 45000. \\
\forall c\,[Car] : PTR(c) = Moderate & \leftarrow 20000 \leq Price(c) \leq 45000 \\
 & \quad \wedge PTR(c) \neq High. \\
\forall c\,[Car] : PTR(c) = Low & \leftarrow Price(c) < 20000 \\
 & \quad \wedge PTR(c) \neq High \\
 & \quad \wedge PTR(c) \neq Moderate.
\end{array}\right\}$$

The rules have a one to one correspondence with the rules in the decision table in Figure 7.2. In other words the given table and this definition represent the same knowledge. All decision tables used to describe the uServ case can in a similar fashion be translated to definitions. An automatic translation from DMN to FO($\cdot$) is still ongoing work.

Since, in general, knowledge represented in the form of decision tables or DMN's boxed expressions can be translated to FO($\cdot$) the inferences present in IDP's inference engine become available for use with DMN decision models. This allows business users to represent the needed knowledge in a business oriented way and leverage the power of generally applicable inferences to enact the modelled decisions.

In Section 7.5 three scenarios are introduced which show the power and usage of several of these inferences using an automatically generated graphical user interface.

7.4 From Decision Modelling to Enactment

As mentioned in Subsection 7.3.2 the rules specified in FO($\cdot$) for the uServ case have a one-to-one correspondence to decision tables and other DMN constructs. As such a translator from DMN to FO($\cdot$) would allow DMN to be used to model business decisions, or other applications where the knowledge is representable in DMN, while IDP can be used as the back-end to perform the required inferences.

7.4.1 Methodology

Building a new application using our approach is a three step process. In a first step the decisions are modelled using DMN. Thus allowing to leverage the understandability of DMN to represent the required knowledge. Translating this DMN decision model to FO(·) forms the second step. Currently this translation has to be done manually, however there is already a simple automated solution for automatically translating the decision tables to RULEML (as indicated in earlier work [125]), this approach can be extended to allow for a translation to FO(·). A direct translation from DMN to FO(·) is currently in progress. After the translation, as a last step, the interface is automatically generated as for the uServ demo. Using IDP's inference engine then allows to enact the decisions in a flexible way.

7.4.2 Architectural Information

The theory editor and interaction with the IDP software is the same as for the IDP Web-IDE described in Chapter 6. The theory editor helps the user with syntax highlighting, some syntactic sugar and useful error messages and warnings.

Apart from the pre-existing IDP and IDE-tools, the application was fully developed using web technologies. Selections of the user are communicated using JSON objects and AngularJS[122] is used to tie the IDP interaction to the web interface.

The tool automatically loads the uServ use case on startup but the tool is completely independent of that. This means that this tool provides the same functionalities for all FO(·) theories and responds immediately to any change. For instance, adding new cars to the high theft probability list, will immediately be reflected in the user interface.

7.4.3 Execution

The IDP system supports the various functionalities through a combination of techniques from SAT, constraint programming and Answer Set Programming. A full description of the system can be found in [36].

The specification is not executable by itself and the functionality which is needed determines how the execution is done. For example, the optimal propagation

grounds the theory and structure to SAT. The approximate version propagates information using symbolic methods and does not do this transformation.

7.5 Enactment Scenarios

This section discusses three enactment scenarios provided by our approach. Firstly direct and indirect consequences of a user's choices are determined, this allows for interactive enactment which guides the user through the decisions. Secondly a partially provided solution can be extended to a full solution. This can be used to generate a possible example, or to determine the outcome of a decision when input is provided. Lastly our approach allows the optimisation of a decision outcome with respect to a given criterion. These scenarios are illustrated with several examples, using a graphical user interface.

The demo's user interface consists of two tabs. The theory tab consists of an editor for the IDP language, which consists of components described in [31]. In this tab an arbitrary FO(.) specification can be placed. In the configuration tab, this specification is converted into an interactive form. It consists of a set of properties, which can have one or multiple values. Every possible value for a property is listed under the property name, together with a "+" and a "-" button. With these buttons, the user can indicate whether this value is true for the property. A "+" indicates the value is true. If neither symbol is selected, it is unknown whether the value is true or not.

This interface allows various functionalities based on the specification in theory tab. These will be explored in the rest of this section.

7.5.1 Determining Consequences

In decisions input and output are strongly related, i.e., even when only partial input is available, information can already be derived. Often it is even possible to exclude certain outcomes, in some cases a unique outcome can already be determined [77]. In other words it is possible that choices made by the user, force some other values to hold, or prevent some other options. In our demo the system communicates this to the user through a green or red background for the respectively forced or impossible values. It also disables the buttons corresponding to that value, thus preventing the user from making inconsistent choices.

In IDP this functionality is provided by the *propagation* inference. This inference tries to find the intersection of all solutions which are still possible, given the

set of current choices. Depending on the size of the problem, this can be done in an exact way, but there are also approximate (more efficient) versions of this inference available for problems where exhaustive search is not feasible.

Figure 7.3 shows the initial configuration of the system. For clarity the examples in the figures have been limited to the decisions involving only cars, and the policy is limited to a single car.

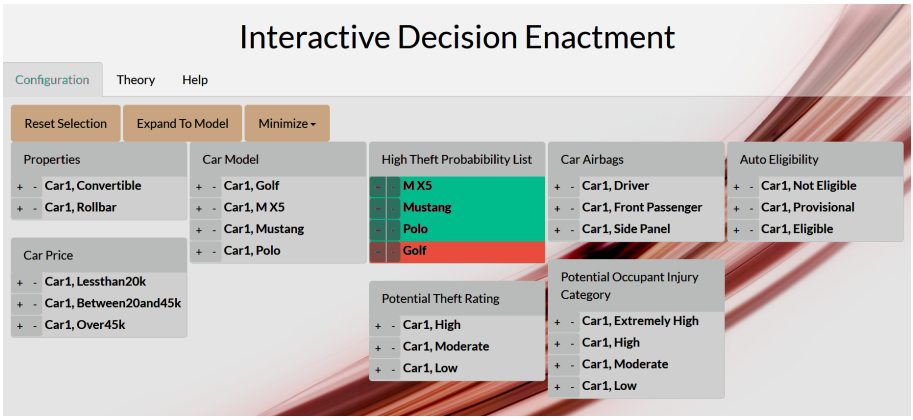

Figure 7.3: Initial situation of the interface for the subset of decision related to cars, in the case of a single car.

Example 7.5.1. When the outcome for the Auto Eligibility decision for `Car1` is selected to be `Eligible` this choice is automatically propagated using the represented knowledge. This will effectively tell the user the minimum requirements for eligibility.

As shown in Figure 7.4 this implies the car must be a `Golf`, since the other models are all on the `High Theft Probability List` and would ensure for the car to get at most a `Provisional` eligibility label. For the same reason the car can not be a convertible of priced over $45 000. Since the `Potential Occupant Injury Risk` of a car can be at most `Moderate` for the car to be `Eligible`, the system can derive the car must have at least `Driver` and `Front Passenger` airbags. ▲

This example shows how a single choice of a desired outcome can have a large effect on the remaining possibilities. Figure 7.4 demonstrates how the extra information about the choices is communicated to the user. Partially provided input can have a similar effect.

Note the difference in background of the '+' sign for `Auto Eligibility(Car1, Eligible)` and that of the propagated information. A coloured background, as

for `Properties(Car1, Convertible)` denote this information is a consequences of the choices that were made. In contrast the clear background of `Car1, Eligible` denotes that this tuple was a choice, i.e., it can be undone by the user. Undoing one of the choices can have an effect on the propagated information. The same equally applies for '-' signs.

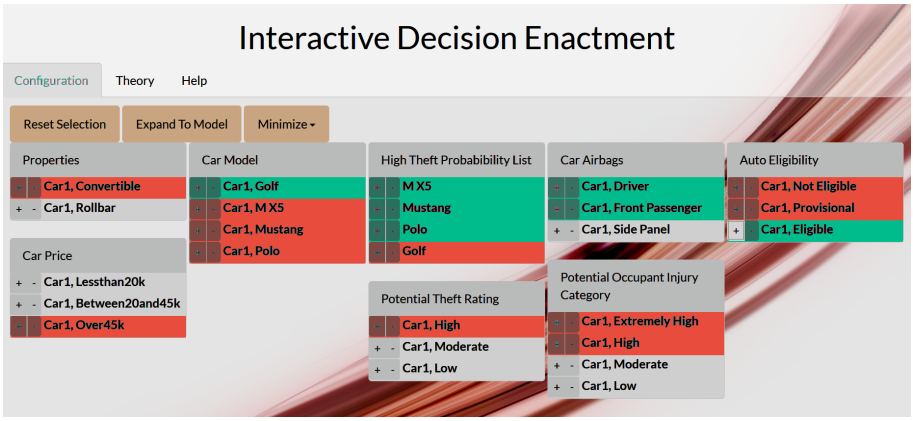

Figure 7.4: Example of Figure 7.3 with the car chosen to be eligible for insurance.

It is important to note that if the inputs for a decision are completely provided, propagation will automatically determine the decision's outcome.

7.5.2 Extending a Partial to a Complete Solution

In some situations one can be interested in a possible solution given the current set of choices, e.g., retaking the previous example, Figure 7.4 can be expanded to a full solution, thus showing a possible situation for an eligible car. This functionality is available in IDP as the *model expansion* inference. Figure 7.5 shows a possible result of the system after a model expansion. Unlike for propagation the information added by expanding the current information to a full solution is not a direct consequence of user choices. Thus all information added by performing *model expansion* has the signs marked by a clear background, i.e., they are choices which can be undone.

7.5.3 Optimisation

Model expansion gives you an arbitrary solution of the problem. Sometimes one can be interested in a particular optimum according to some criterion such

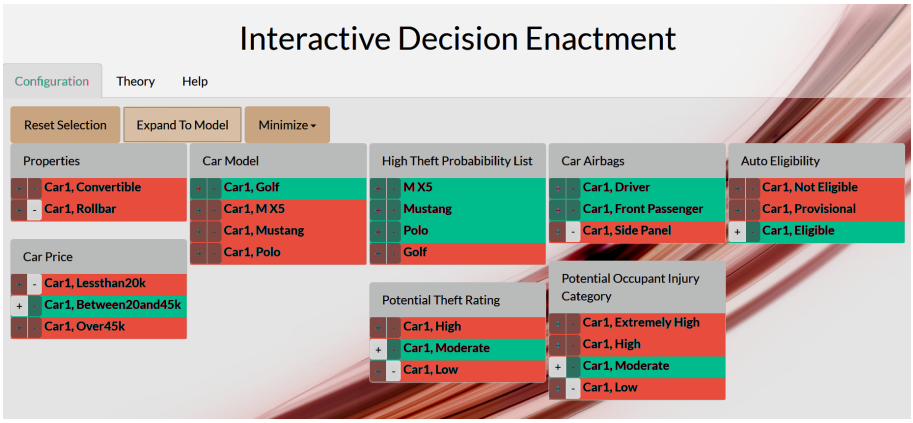

Figure 7.5: Example of Figure 7.4 after expanding the selection to a model.

as cost or time. For example in the situation where you want to buy a car, and want to know what car to buy so your insurance policy is as cheap as possible.

The inference needed for solving optimization problems like this is also provided by IDP. The possible optimisation criteria defined in the specification are listed under the drop down menu "Optimize". The use of this inference is shown in Example 7.5.2. All rules up to determining the policy's eligibility are used. Additionally, to allow for a meaningful use case, the example entails two drivers and three cars.

Example 7.5.2. The optimisation inference can for example be used to enact the eligibility decision for a customer asking for a policy covering as many of her cars as possible as well as preferably covering her grand son, however he has been convicted of a DUI in the past. She owns three cars, an `MX5`, a `Golf`, and a `Polo`. The Polo does not have driver side airbags. The policy should not be immediately rejected, however requiring an underwrite would not be a problem. The customer is not an `elite`, `preferred`, or `long term client`. After filling in all these details the optimize inference with the `Max Drivers Cars` term can be used to infer the most optimal policy.

The resulting policy covers both the customer and grandson, and two of her three cars. Using the interface it is also possible to see what would be the result if the grandson was not covered by the insurance policy, by deselecting him from the `Covers Driver` predicate. In this case, the third car can also be covered by the policy. This leaves the customer with two choices, which are equivalent according to the criterion. ▲

This inference can be used with any user defined term, similar to the `Max Drivers Cars` term. This term represents the sum of the amount of drivers and cars covered by the insurance policy. Since internally the minimize inference is used, the term is negated to allow for maximization. The `Max Drivers Cars` term is specified in IDP as follows.

```
term MaxDriversCars : V {
- (#{d : CoversDriver(d)} + #{c : CoversCar(c)})
}
```

7.6 Major Contributions and Future Work

The power of our proposed approach is in the availability of multiple forms of inference. The transformation from DMN to FO($\cdot$) allows the flexible reasoning, available in IDP, to be used for DMN models. In particular this means the same DMN decision model can be used for different scenarios. The demo explained in Section 7.5 shows only the propagation, model expansion, and minimization inference. However the other inferences available in the IDP system can equally be useful for business applications.

Additionally derived inferences can be implemented specifically geared towards decision management. Inferences can be defined which check the completeness and exclusivity of a decision. This corresponds to respectively determining that there is at least one output for each input and at most one output for each input. By using *model expansion* as a base for these inferences counter examples can be generated in case the decision in question is not complete or exclusive. This can be done by not specifying the input and looking for a model in which there is no output or multiple outputs. These are models which should not be models and can help in detecting where the violations occur, and aid in resolving them.

The graphical user interface provided for the demo, shows exactly one outcome of a decision enactment. However in many cases alternative outcome may be possible. Especially in the case of only partially provided input information, there may not be a unique outcome. The *model expansion* inference can be used to determine all possible outcome alternatives, allowing the user to choose its most preferable model. This extends to the use of the *optimize* inference. In Example 7.5.2 two outcomes are possible, one covering the grandson, and another covering the third car. Further work is needed to explore what the best ways are to present these options to the user.

The interface offers a way to show options, user choices and their consequences in a general manner. Currently it displays all possibilities in the form of a check

box for true and false. For larger domains and applications this may become cumbersome. In the future the interface will be extended to allow the efficient visualisation of larger domains, as well as allow for more clear modularisation of the decisions. This would allow to include the price of the insurance policy in the modelling, e.g., allowing it to be used with the optimisation inferences to minimize the price of the policy.

Since the original publication of the decision enactment tool, a second version of this tool has been developed including some of the more advanced features described above. Since then, we developed new applications using combining DMN and the knowledge base paradigm. The most important application has been done in the context of Machine Component Design[3], where our tool enabled a company to interactively explore the design space of the products they can produce.

7.7 Conclusion

Enacting business decisions is challenging, especially for an expressive decision modelling language, such as DMN. Our approach uses the knowledge base system IDP as an inference engine for DMN. Through translation from DMN to an extension of first-order logic, we can leverage the flexibility available in IDP. Additionally a graphical user interface is provided which is automatically generated from the translated modelling. This ensures that our approach is generally applicable. We presented our work using the well-known uServ product derby case study as a test case. The examples show our approach is able to enact decisions in several ways. It guides the user by immediately showing the consequences of the choices that are made, it allows a partial solution to be expanded to a total solution, and it allows to ask for an optimal solution with respect to a chosen criterion.

Our approach consists of three steps. The decision knowledge is modelled using DMN. This modelling is translated to an extension of first-order logic, FO($\cdot$). The graphical user interface is automatically generated from this representation, and uses IDP's inference engine to enact the decisions. Several extensions to this approach are mentioned as future work. The user interface can be extended to allow for more complex data types. The automation of the translation from DMN to FO($\cdot$), which is currently done manually, is also left as future work. New, derived inferences specific to decision enactment will be devised to provide more possibilities to use the represented decisions.

8 Conclusion

This thesis investigated possibilities for new higher-order modelling languages. One important motivation is that through the use of higher-order modelling languages it can become possible to obtain modellings which are more succinct and readable without compromising efficiency of an implementation which does inferences on these languages. To reach this goal, a lot of attention was given to the lambda calculus, in which there is no distinction between first-order and higher-order concepts.

Part 1 In the first part of the thesis we focused on the semantical problems of defining higher-order languages. Chapter 2 identifies templates as second-order definitions. This enabled us to give a semantics to templates without needing to resort to an algorithmic rewriting system. Chapter 3 presents a logic based on the simply typed lambda calculus but fully integrated with definitions under the well-founded semantics.

This part also contains inspiration for continued research. A tighter integration of functions and inductive definitions could lead to a more elegant language. Additionally, there is still a gap between what is efficiently implementable and the logics presented in the first part.

Part 2 In the second part of the thesis a more practical approach was taken. The focus lays on how these higher-order languages can be translated to a more traditional modelling language: Answer Set Programming. Chapter 4 explained how a prototypical lambda calculus, Programming Computable Functions, can be translated to ASP. It also explained how this translation can be extended for extensions of PCF. Chapter 5 showed how a full system for model expansion can be built using these principles.

More research continuing the work of the second part could investigate the integration of more traditional grounding optimisations with the new language constructs and implementation. A look into the efficiency of the prototype system is also an important next step. This could lead to the identification of bottle necks in the system and more insight in the advantages and disadvantages of the approach taken.

Part 3 The last part of the thesis was even more practical and looked at tooling for the IDP3 system. These tools aim to make it easier to use IDP3. Chapter 6 introduced a web-based IDE. It encompasses the standard features of an IDE and the web interface allows people to try IDP without actually installing it. Chapter 7 shows a tool in which it is possible to interactively explore the solution space of a theory. It was built to show how Decision Modelling and Notation can be used for more than drawing conclusions from some premises, however this tool can also be used in a wide range of interactive configuration problems.

Both tools leave ample room for improvement. The IDE does not do any semantical analysis to help the user. Such an analysis would allow for more user guidance while writing theories. Improvements on the tool presented in Chapter 7 are under development. The biggest improvement in the new tool is the support for explanations of propagations.

A

Defining Letrec in terms of Fixpoints

In this appendix we will more formally define the semantics of letrecs with mutually recursive bindings. This problem was posed in Section 4.6.2.

Example A.0.1. Consider the functions `even` and `odd`. These are mutually recursively defined. We use 0 to denote true and 1 to denote false.

```
letrec even = λi. ifz i then 0 else odd (pred i)
       odd  = λi. ifz i then 1 else even (pred i)
in (even 3)
```

▲

The goal is to rewrite letrec expressions to a form in which there is no letrec, but we can use fixpoints to obtain recursion. We will use the same trick as for letrec expressions with a single binding to abstract away the bound variables. Consider a general letrec, it takes the form:

```
letrec x₁ = a₁
       ...
       xₙ = aₙ
in f
```

We will rewrite this to: `(λx₁. ... λxₙ. f) b₁ ... bₙ.`

Example A.0.2. We are looking for expressions of $\texttt{even}_{fix}$ and $\texttt{odd}_{fix}$ such that `(λeven. λodd. even 3)` $\texttt{even}_{fix}$ $\texttt{odd}_{fix}$ evaluates to one. ▲

This leaves us to define the expressions of the form $\texttt{b}_i$. The fixpoint operation which is built into the language fixes a single argument which only works for self-recursion. We will now introduce a way to weave this regular fixpoint between expressions so that a fixpoint can be reached which respects the mutual recursion.

In the derivation of the expressions we will need to have a fixpoint functions for the first argument of a function of higher arity. For a function of arity `i` this modified fixpoint function can easily be defined in PCF like this:

$\texttt{arfix}_i = \lambda \texttt{f}.\ \lambda \texttt{x}_2.\ \lambda \texttt{x}_3\ \ldots\ \lambda \texttt{x}_n.\ \texttt{fix}\ (\lambda \texttt{x}_1.\ \texttt{f}\ \texttt{x}_1\ \ldots\ \texttt{x}_n)$

We will use expressions of the form $\texttt{e}_{i,j}$, the `i`-index represents the number of the bound variable, the `j`-index represents how many bound variables are fixed. So $\texttt{e}_{i,j}$ represents a definition of $\texttt{x}_i$ in which the first `j` variables are fixed, and thus `n-j` variables still need to be fixed. So our goal is that $\texttt{b}_i = \texttt{e}_{i,n}$.

The definitions of the $\texttt{e}_{i,0}$ is straightforward, as no fixpoints are needed for this one. This just represents the binding where all recursion is abstracted. Fixing a new level can be done through the `arfix` function in case `i = j`. Whenever $\texttt{i} \neq \texttt{j}$ we need to refer to the version where `i = j` to properly define $\texttt{e}_{i,j}$.

$$\texttt{e}_{i,0} = \lambda \texttt{x}_1.\ \ldots\ \lambda \texttt{x}_n.\ \texttt{a}_i$$
$$\texttt{e}_{i,i} = \texttt{arfix}_{n-i+1}\ \texttt{e}_{i,i-1}$$
$$\texttt{e}_{i,j} = \lambda \texttt{x}_{j+1}.\ \ldots\ \lambda \texttt{x}_n.\ \texttt{e}_{i,j-1}\ (\texttt{e}_{i,i}\ \texttt{x}_{j+1}\ \ldots\ \texttt{x}_n)\ \texttt{x}_{j+1}\ \ldots\ \texttt{x}_n \quad (\text{if } i \neq j)$$

Note that there is a full stratification of these $\texttt{e}_{i,j}$. So the full $\texttt{e}_{i,n}$ can be constructed. Some subterms are used multiple times, so a more readable format for these terms can use nested non-recursive let constructs.

Using this set of definitions we can properly add letrec expressions to the semantics of PCF.

$$\frac{E,\ (\lambda \texttt{x}_1.\ \ldots \lambda \texttt{x}_n.\ \texttt{f})\ \texttt{e}_{1,n}\ \ldots\ \texttt{e}_{n,n} \Downarrow v}{E,\ \texttt{letrec}\ \texttt{x}_1 = \texttt{a}_1\ ;\ \ldots\ ;\ \texttt{x}_n = \texttt{a}_n\ \texttt{in}\ \texttt{f} \Downarrow v}$$

Example A.0.3. The odd/even example can be rewritten using the technique described above. The different $\texttt{e}_i$ look as follows:

$$\texttt{even}_0 = \lambda\texttt{even}.\lambda\texttt{odd}.\lambda\texttt{i}.\ \texttt{ifz i then 0 else odd (pred i)}$$
$$\texttt{odd}_0 = \lambda\texttt{even}.\lambda\texttt{odd}.\lambda\texttt{i}.\ \texttt{ifz i then 1 else even (pred i)}$$
$$\texttt{even}_1 = \lambda\texttt{odd}.\ \texttt{fix}\ (\lambda\texttt{even}.\ \texttt{even}_0\ \texttt{even odd})$$
$$\texttt{odd}_1 = \lambda\texttt{odd}.\ \texttt{odd}_0\ (\texttt{even}_1\ \texttt{odd})\ \texttt{odd}$$
$$\texttt{even}_2 = \texttt{even}_1\ \texttt{odd}_2$$
$$\texttt{odd}_2 = \texttt{fix odd}_1$$

▲

Example A.0.3 follows the schema in a strict way. However, this can be optimised for more practical applications. In the base cases, there is no need to abstract away the variables which are unused in the body of the binding. Using this optimisation makes the rewriting more readable and more efficient in implementations.

Example A.0.4. By binding only the variables which are actually used, Example A.0.3 can be simplified to:

```
even₀ = λodd.λi. ifz i then 0 else odd (pred i)
odd₀  = λeven.λi. ifz i then 1 else even (pred i)
even₂ = fix (λeven. even₀ (odd₀ even))
odd₂  = odd₀ even₂
```

▲

Extending the translation relation of Chapter 4 is simple. As the rewriting step shown in the semantics never introduces a new letrec. A letrec can simply be translated by doing the rewriting step and translating the rewritten expression.

$$\frac{E, \texttt{(}\lambda\texttt{x}_1\texttt{. ...}\lambda\texttt{x}_n\texttt{. f) e}_{1,n}\texttt{ ... e}_{n,n} \Downarrow v}{E, \texttt{letrec x}_1\texttt{ = a}_1\texttt{ ; ... ; x}_n\texttt{ = a}_n\texttt{ in f} \Downarrow v}$$

Bibliography

[1] Many faces of the fixed-point combinator. `http://okmij.org/ftp/Computation/fixed-point-combinators.html#Poly-variadic`, 2013.

[2] Abramson, H., and Rogers, M. H. *Meta-programming in logic programming.* MIT Press, 1989.

[3] Aerts, B., and Vennekens, J. Application of Logic-Based Methods to Machine Component Design. In *Technical Communications of the 34th International Conference on Logic Programming (ICLP 2018)* (Dagstuhl, Germany, 2018), A. D. Palu', P. Tarau, N. Saeedloei, and P. Fodor, Eds., vol. 64 of *OpenAccess Series in Informatics (OASIcs)*, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, pp. 13:1–13:15.

[4] Appel, A. W. *Modern Compiler Implementation in C.* Cambridge University Press, 1998.

[5] Apt, K. R. *Principles of Constraint Programming.* Cambridge University Press, 2003.

[6] Atom.io. `https://atom.io/`. Accessed: 2015-07-01.

[7] Baral, C. *Knowledge Representation, Reasoning, and Declarative Problem Solving.* Cambridge University Press, New York, NY, USA, 2003.

[8] Baral, C., Dzifcak, J., and Takahashi, H. Macros, macro calls and use of ensembles in modular answer set programming. In *Logic Programming, 22nd International Conference, ICLP 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings* (2006), S. Etalle and M. Truszczynski, Eds., vol. 4079 of *Lecture Notes in Computer Science*, Springer, pp. 376–390.

[9] Barwise, J., and Etchemendy, J. Information, infons, and inference. *Situation theory and its applications 1*, 22 (1990).

[10] Biere, A., Heule, M., van Maaren, H., and Walsh, T., Eds. *Handbook of Satisfiability* (2009), vol. 185 of *Frontiers in Artificial Intelligence and Applications*, IOS Press.

[11] Blockeel, H., Bogaerts, B., Bruynooghe, M., De Cat, B., De Pooter, S., Denecker, M., Labarre, A., Ramon, J., and Verwer, S. Modeling machine learning and data mining problems with FO(·). In Dovier and Santos Costa [46], pp. 14–25.

[12] Bogaerts, B. *Groundedness in logics with a fixpoint semantics.* PhD thesis, Department of Computer Science, KU Leuven, June 2015. Denecker, Marc (supervisor), Vennekens, Joost and Van den Bussche, Jan (cosupervisors).

[13] Bogaerts, B., Jansen, J., Bruynooghe, M., De Cat, B., Vennekens, J., and Denecker, M. Simulating dynamic systems using linear time calculus theories. *TPLP 14*, 4–5 (7 2014), 477–492.

[14] Bootstrap. `http://getbootstrap.com/`. Accessed: 2015-07-01.

[15] Brewka, G., Eiter, T., and Truszczynski, M. Answer set programming at a glance. *Commun. ACM 54*, 12 (2011), 92–103.

[16] Bruynooghe, M., Blockeel, H., Bogaerts, B., De Cat, B., De Pooter, S., Jansen, J., Labarre, A., Ramon, J., Denecker, M., and Verwer, S. Predicate logic as a modeling language: modeling and solving some machine learning and data mining problems with IDP3. *TPLP 15*, 6 (November 2015), 783–817.

[17] Calimeri, F., Faber, W., Gebser, M., Ianni, G., Kaminski, R., Krennwallner, T., Leone, N., Ricca, F., and Schaub, T. ASP-Core-2 input language format. Tech. rep., ASP Standardization Working Group, 2013.

[18] Calimeri, F., and Ianni, G. Template programs for disjunctive logic programming: An operational semantics. *AI Commun. 19*, 3 (2006), 193–206.

[19] Charalambidis, A., Rondogiannis, P., and Symeonidou, I. Approximation fixpoint theory and the well-founded semantics of higher-order logic programs. *TPLP 18*, 3-4 (2018), 421–437.

[20] Chen, W., Kifer, M., and Warren, D. S. Hilog: A foundation for higher-order logic programming. *The Journal of Logic Programming 15*, 3 (1993), 187–230.

[21] Chen, W., Kifer, M., and Warren, D. S. Hilog: A foundation for higher-order logic programming. *The Journal of Logic Programming 15*, 3 (1993), 187–230.

[22] Clément, D., Despeyroux, J., Despeyroux, T., and Kahn, G. A simple applicative language: Mini-ml. In *LISP and Functional Programming* (1986), pp. 13–27.

[23] Cliffe, O., Vos, M. D., Brain, M., and Padget, J. A. ASPVIZ: declarative visualisation and animation using answer set programming. In *Logic Programming, 24th International Conference, ICLP 2008, Udine, Italy, December 9-13 2008, Proceedings* (2008), M. G. de la Banda and E. Pontelli, Eds., vol. 5366 of *Lecture Notes in Computer Science*, Springer, pp. 724–728.

[24] Cloud9 - your development environment, in the cloud. `https://c9.io/`. Accessed: 2015-07-01.

[25] Codemirror. `http://www.codemirror.net`. Accessed: 2015-07-01.

[26] Codiad. `http://codiad.com/`. Accessed: 2015-07-01.

[27] Damaggio, E., Deutsch, A., Hull, R., and Vianu, V. Automatic verification of data-centric business processes. In *Business Process Management - 9th International Conference, BPM 2011, Clermont-Ferrand, France, August 30 - September 2, 2011. Proceedings* (2011), S. Rinderle-Ma, F. Toumani, and K. Wolf, Eds., vol. 6896 of *Lecture Notes in Computer Science*, Springer, pp. 3–16.

[28] Dao-Tran, M., Eiter, T., Fink, M., and Krennwallner, T. Modular nonmonotonic logic programming revisited. In *Logic Programming, 25th International Conference, ICLP 2009, Pasadena, CA, USA, July 14-17, 2009. Proceedings* (2009), P. M. Hill and D. S. Warren, Eds., vol. 5649 of *Lecture Notes in Computer Science*, Springer, pp. 145–159.

[29] Dargaye, Z., and Leroy, X. Mechanized verification of CPS transformations. In *Logic for Programming, Artificial Intelligence, and Reasoning, 14th International Conference, LPAR 2007, Yerevan, Armenia, October 15-19, 2007, Proceedings* (2007), N. Dershowitz and A. Voronkov, Eds., vol. 4790 of *Lecture Notes in Computer Science*, Springer, pp. 211–225.

[30] Dasseville, I., and Denecker, M. Transpiling programming computable functions to answer set programs. In Silva [113], pp. 3–17.

[31] Dasseville, I., and Janssens, G. A web-based IDE for IDP. *CoRR abs/1511.00920* (2015).

[32] Dasseville, I., and Janssens, G. FMS: functional programming as a modelling language. In Silva [113], pp. 145–161.

[33] Dasseville, I., Janssens, L., Janssens, G., Vanthienen, J., and Denecker, M. Combining DMN and the knowledge base paradigm for flexible decision enactment. In *Supplementary Proceedings of the RuleML 2016 Challenge, Doctoral Consortium and Industry Track hosted by the 10th International Web Rule Symposium, RuleML 2016, New York, USA, July 6-9, 2016.* (2016), T. Athan, A. Giurca, R. Grütter, M. Proctor, K. Teymourian, and W. V. Woensel, Eds., vol. 1620 of *CEUR Workshop Proceedings*, CEUR-WS.org.

[34] Dasseville, I., van der Hallen, M., Bogaerts, B., Janssens, G., and Denecker, M. A Compositional Typed Higher-Order Logic with Definitions. In *Technical Communications of the 32nd International Conference on Logic Programming (ICLP 2016)* (Dagstuhl, Germany, 2016), M. Carro, A. King, N. Saeedloei, and M. D. Vos, Eds., vol. 52 of *OpenAccess Series in Informatics (OASIcs)*, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, pp. 14:1–14:13.

[35] Dasseville, I., van der Hallen, M., Janssens, G., and Denecker, M. Semantics of templates in a compositional framework for building logics. *TPLP 15*, 4–5 (2015), 681–695.

[36] De Cat, B. *Separating Knowledge from Computation: An FO(·) Knowledge Base System and its Model Expansion Inference.* PhD thesis, KU Leuven, Leuven, Belgium, May 2014.

[37] De Cat, B., Bogaerts, B., Bruynooghe, M., Janssens, G., and Denecker, M. Predicate logic as a modelling language: The IDP system. *CoRR abs/1401.6312v2* (2016).

[38] Denecker, M. Extending classical logic with inductive definitions. In *CL* (2000), J. W. Lloyd, V. Dahl, U. Furbach, M. Kerber, K.-K. Lau, C. Palamidessi, L. M. Pereira, Y. Sagiv, and P. J. Stuckey, Eds., vol. 1861 of *LNCS*, Springer, pp. 703–717.

[39] Denecker, M., Lierler, Y., Truszczyński, M., and Vennekens, J. A Tarskian informal semantics for answer set programming. In Dovier and Costa [46], pp. 277–289.

[40] Denecker, M., Marek, V., and Truszczyński, M. Approximations, stable operators, well-founded fixpoints and applications in nonmonotonic

reasoning. In *Logic-Based Artificial Intelligence*, J. Minker, Ed., vol. 597 of *The Springer International Series in Engineering and Computer Science*. Springer US, 2000, pp. 127–144.

[41] Denecker, M., Marek, V. W., and Truszczynski, M. Ultimate approximation and its application in nonmonotonic knowledge representation systems. *Inf. Comput. 192*, 1 (2004), 84–121.

[42] Denecker, M., and Ternovska, E. A logic of nonmonotone inductive definitions. *ACM Trans. Comput. Log. 9*, 2 (Apr. 2008), 14:1–14:52.

[43] Denecker, M., and Vennekens, J. Building a knowledge base system for an integration of logic programming and classical logic. In *ICLP* (2008), M. García de la Banda and E. Pontelli, Eds., vol. 5366 of *LNCS*, Springer, pp. 71–76.

[44] Denecker, M., and Vennekens, J. The well-founded semantics is the principle of inductive definition, revisited. In *KR* (2014), C. Baral, G. De Giacomo, and T. Eiter, Eds., AAAI Press, pp. 1–10.

[45] Devlin, K. *Logic and information*. Cambridge University Press, 1991.

[46] Dovier, A., and Costa, V. S., Eds. *Technical Communications of the 28th International Conference on Logic Programming, ICLP 2012, September 4-8, 2012, Budapest, Hungary. Proceedings* (2012), vol. 17 of *LIPIcs*, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik.

[47] Dowek, G., and Lévy, J. *Introduction to the Theory of Programming Languages*. Undergraduate Topics in Computer Science. Springer, 2011.

[48] Eclipse plugin development environment. `https://eclipse.org/pde/`. Accessed: 2015-07-01.

[49] Eiter, T., Fink, M., Ianni, G., Krennwallner, T., Redl, C., and Schüller, P. A model building framework for answer set programming with external computations. *TPLP 16*, 4 (2016), 418–464.

[50] Eiter, T., Ianni, G., Schindlauer, R., and Tompits, H. A uniform integration of higher-order reasoning and external evaluations in answer-set programming. In *IJCAI-05, Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence, Edinburgh, Scotland, UK, July 30 - August 5, 2005* (2005), L. P. Kaelbling and A. Saffiotti, Eds., Professional Book Center, pp. 90–96.

[51] Eiter, T., Krennwallner, T., and Redl, C. Hex-programs with nested program calls. In *Applications of Declarative Programming and*

Knowledge Management - 19th International Conference, INAP 2011, and 25th Workshop on Logic Programming, WLP 2011, Vienna, Austria, September 28-30, 2011, Revised Selected Papers (2011), H. Tompits, S. Abreu, J. Oetsch, J. Pührer, D. Seipel, M. Umeda, and A. Wolf, Eds., vol. 7773 of *Lecture Notes in Computer Science*, Springer, pp. 269–278.

[52] Faber, W., Pfeifer, G., and Leone, N. Semantics and complexity of recursive aggregates in answer set programming. *Artif. Intell. 175*, 1 (2011), 278–298.

[53] Febbraro, O., Reale, K., and Ricca, F. ASPIDE: integrated development environment for answer set programming. In *Logic Programming and Nonmonotonic Reasoning - 11th International Conference, LPNMR 2011, Vancouver, Canada, May 16-19, 2011. Proceedings* (2011), J. P. Delgrande and W. Faber, Eds., vol. 6645 of *Lecture Notes in Computer Science*, Springer, pp. 317–330.

[54] Feldman, J., and Taylor, J. Good old userv product derby in the brave new world of decision management. In *Building Business Capability 2015*. 2015.

[55] Ferraris, P. Answer sets for propositional theories. In *Proceedings of International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)* (2005), pp. 119–131.

[56] Fourer, R., Gay, D. M., and Kernighan, B. W. Ampl: A mathematical programming language.

[57] Friedman-Hill, E. *Jess in action: Rule-based systems in Java.* In Action series. Manning, 2003.

[58] Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T., and Thiele, S. A user's guide to gringo, clasp, clingo, and iclingo. `http://downloads.sourceforge.net/potassco/guide.pdf`, 2010.

[59] Gebser, M., Kaminski, R., Kaufmann, B., and Schaub, T. Clingo = ASP + control: Preliminary report. *CoRR abs/1405.3694* (2014).

[60] Gelfond, M., and Zhang, Y. Vicious circle principle and logic programs with aggregates. *TPLP 14*, 4–5 (2014), 587–601.

[61] Germano, S., Calimeri, F., and Palermiti, E. Loide: A web-based IDE for logic programming preliminary report. In *Practical Aspects of Declarative Languages - 20th International Symposium, PADL 2018, Los Angeles, CA, USA, January 8-9, 2018, Proceedings* (2018), F. Calimeri, K. W. Hamlen, and N. Leone, Eds., vol. 10702 of *Lecture Notes in Computer Science*, Springer, pp. 152–160.

[62] Giordano, L., Martelli, A., Spiotta, M., and Dupré, D. T. Business process verification with constraint temporal answer set programming. *Theory and Practice of Logic Programming 13*, 4-5 (2013), 641–655.

[63] About Gists. `https://help.github.com/articles/about-gists/`. Accessed: 2015-07-01.

[64] Goedertier, S., Mues, C., and Vanthienen, J. Specifying process-aware access control rules in sbvr. In *Advances in Rule Interchange and Applications* (2007), Springer, pp. 39–52.

[65] Governatori, G. Representing business contracts in *RuleML. Int. J. Cooperative Inf. Syst. 14*, 2-3 (2005), 181–216.

[66] Gratzer, D. A Tiny Compiler For A Typed Higher Order Language, 2015.

[67] Group, O. M. Semantics of business vocabulary and business rules (sbvr). OMG document number formal/08-01-02, January 2008. Version 1.0.

[68] Guns, T. Declarative pattern mining using constraint programming. *Constraints 20*, 4 (2015), 492–493.

[69] Gurevich, Y., and Neeman, I. Logic of infons: The propositional case. *ACM Trans. Comput. Log. 12*, 2 (2011), 9.

[70] Hertum, P. V., Dasseville, I., Janssens, G., and Denecker, M. The KB paradigm and its application to interactive configuration. In *Practical Aspects of Declarative Languages - 18th International Symposium, PADL 2016, St. Petersburg, FL, USA, January 18-19, 2016. Proceedings* (2016), M. Gavanelli and J. H. Reppy, Eds., vol. 9585 of *Lecture Notes in Computer Science*, Springer, pp. 13–29.

[71] Hobbs, J. R., and Rosenschein, S. J. Making computational sense of montague's intensional logic. *Artificial Intelligence 9*, 3 (1977), 287–306.

[72] Icecoder. `https://icecoder.net/`. Accessed: 2015-07-01.

[73] IDPDraw: Finite structure visualization. `http://dtai.cs.kuleuven.be/krr/software/visualisation`, 2012.

[74] IDP-IDE: An integrated development environment for IDP. `https://dtai.cs.kuleuven.be/krr/software/idp-ide`, 2013.

[75] Iwata, S., and Kasai, T. The othello game on an n*n board is pspace-complete. *Theor. Comput. Sci. 123*, 2 (1994), 329–340.

[76] Jansen, J., Jorissen, A., and Janssens, G. Compiling input* FO(·) inductive definitions into tabled Prolog rules for IDP3. *TPLP 13*, 4–5 (2013), 691–704.

[77] Janssens, L., De Smedt, J., and Vanthienen, J. Modeling and enacting enterprise decisions. In *Advanced Information Systems Engineering Workshops* (2016), Springer. Accepted.

[78] Jones, S. L. P., and Marlow, S. Secrets of the Glasgow Haskell Compiler inliner. *J. Funct. Program. 12*, 4&5 (2002), 393–433.

[79] Kaufmann, B., Leone, N., Perri, S., and Schaub, T. Grounding and solving in answer set programming. *AI Magazine 37*, 3 (2016), 25–32.

[80] Kelsey, R., Clinger, W. D., and Rees, J. Revised^5 report on the algorithmic language scheme. *SIGPLAN Notices 33*, 9 (1998), 26–76.

[81] Kemp, D. B., and Stuckey, P. J. Semantics of logic programs with aggregates. In *ISLP* (1991), V. A. Saraswat and K. Ueda, Eds., MIT Press, pp. 387–401.

[82] Kleene, S. C. *Introduction to Metamathematics*. Van Nostrand, 1952.

[83] Kloimüllner, C., Oetsch, J., Pührer, J., and Tompits, H. Kara: A system for visualising and visual editing of interpretations for answer-set programs. In *Applications of Declarative Programming and Knowledge Management - 19th International Conference, INAP 2011, and 25th Workshop on Logic Programming, WLP 2011, Vienna, Austria, September 28-30, 2011, Revised Selected Papers* (2011), H. Tompits, S. Abreu, J. Oetsch, J. Pührer, D. Seipel, M. Umeda, and A. Wolf, Eds., vol. 7773 of *Lecture Notes in Computer Science*, Springer, pp. 325–344.

[84] Lapauw, R., Dasseville, I., and Denecker, M. Visualising interactive inferences with IDPD3. In *International Workshop on User-Oriented Logic Programming, IULP 2015, Cork, Ireland, August 31, 2015. Proceedings* (2015). Accepted.

[85] Leach, J., Nieva, S., and Rodríguez-Artalejo, M. Constraint logic programming with hereditary harrop formula. *CoRR cs.PL/0404053* (2004).

[86] Leuschel, M., and Butler, M. ProB: A model checker for B. In *FME 2003: Formal Methods* (2003), K. Araki, S. Gnesi, and D. Mandrioli, Eds., LNCS 2805, Springer-Verlag, pp. 855–874.

[87] Lifschitz, V. Answer set planning. In *Logic Programming: The 1999 International Conference, Las Cruces, New Mexico, USA, November 29 - December 4, 1999* (1999), D. De Schreye, Ed., MIT Press, pp. 23–37.

[88] Marcopoulos, E., Reotutar, C., and Zhang, Y. An online development environment for answer set programming. *CoRR abs/1707.01865* (2017).

[89] Marek, V., and Truszczyński, M. Stable models and an alternative logic programming paradigm. In *The Logic Programming Paradigm: A 25-Year Perspective*, K. R. Apt, V. Marek, M. Truszczyński, and D. S. Warren, Eds. Springer-Verlag, 1999, pp. 375–398.

[90] Marlow, S., Peyton Jones, S., Kmett, E., and Mokhov, A. Desugaring haskell's do-notation into applicative operations. In *Proceedings of the 9th International Symposium on Haskell* (2016), ACM, pp. 92–104.

[91] Microsoft. Language Server Protocol, 2018.

[92] Mitchell, J. C. On abstraction and the expressive power of programming languages. *Sci. Comput. Program. 21*, 2 (1993), 141–163.

[93] Mumick, I. S., Finkelstein, S. J., Pirahesh, H., and Ramakrishnan, R. Magic is relevant. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, Atlantic City, NJ, May 23-25, 1990.* (1990), H. Garcia-Molina and H. V. Jagadish, Eds., ACM Press, pp. 247–258.

[94] Musser, D. R., Derge, G. J., and Saini, A. *STL tutorial and reference guide: C++ programming with the standard template library.* Addison-Wesley Professional, 2009.

[95] Nadathur, G., and Miller, D. An overview of LambdaProlog. In *Fifth International Conference and Symposium on Logic Programming. MIT Press* (1988).

[96] Nagl, C., Rosenberg, F., and Dustdar, S. Vidre–a distributed service-oriented business rule engine based on ruleml. In *Enterprise Distributed Object Computing Conference, 2006. EDOC'06. 10th IEEE International* (2006), IEEE, pp. 35–44.

[97] Nethercote, N., Stuckey, P. J., Becket, R., Brand, S., Duck, G. J., and Tack, G. Minizinc: Towards a standard CP modelling language. In *CP'07* (2007), C. Bessiere, Ed., vol. 4741 of *LNCS*, Springer, pp. 529–543.

[98] Niemelä, I. Logic programs with stable model semantics as a constraint programming paradigm. *Ann. Math. Artif. Intell. 25*, 3-4 (1999), 241–273.

[99] Niemelä, I. Answer set programming: A declarative approach to solving search problems. In *JELIA* (2006), M. Fisher, W. van der Hoek, B. Konev, and A. Lisitsa, Eds., vol. 4160 of *LNCS*, Springer, pp. 15–18. Invited talk.

[100] Nigam, A., and Caswell, N. S. Business artifacts: An approach to operational specification. *IBM Systems Journal 42*, 3 (2003), 428–445.

[101] Oetsch, J., Pührer, J., Seidl, M., Tompits, H., and Zwickl, P. VIDEAS: A development tool for answer-set programs based on model-driven engineering technology. In *Logic Programming and Nonmonotonic Reasoning - 11th International Conference, LPNMR 2011, Vancouver, Canada, May 16-19, 2011. Proceedings* (2011), J. P. Delgrande and W. Faber, Eds., vol. 6645 of *Lecture Notes in Computer Science*, Springer, pp. 382–387.

[102] Oetsch, J., Pührer, J., and Tompits, H. The SeaLion has landed: An IDE for answer-set programming - preliminary report. In *Applications of Declarative Programming and Knowledge Management - 19th International Conference, INAP 2011, and 25th Workshop on Logic Programming, WLP 2011, Vienna, Austria, September 28-30, 2011, Revised Selected Papers* (2011), H. Tompits, S. Abreu, J. Oetsch, J. Pührer, D. Seipel, M. Umeda, and A. Wolf, Eds., vol. 7773 of *Lecture Notes in Computer Science*, Springer, pp. 305–324.

[103] OMG. Decision Model and Notation, 2015. Accessed on 17.05.2016.

[104] Parr, T. *The definitive ANTLR 4 reference*. Pragmatic Bookshelf, 2013.

[105] Pelov, N., Denecker, M., and Bruynooghe, M. Well-founded and stable semantics of logic programs with aggregates. *TPLP 7*, 3 (2007), 301–353.

[106] Pierce, B. C. *Types and Programming Languages*. MIT Press, Cambridge, MA, USA, 2002.

[107] Plotkin, G. D. LCF considered as a programming language. *Theor. Comput. Sci. 5*, 3 (1977), 223–255.

[108] Provetti, A., and Son, T. C., Eds. *Answer Set Programming, Towards Efficient and Scalable Knowledge Representation and Reasoning, Proceedings of the 1st Intl. ASP'01 Workshop, Stanford, March 26-28, 2001* (2001).

[109] Ricca, F., Gallucci, L., Schindlauer, R., Dell'Armi, T., Grasso, G., and Leone, N. OntoDLV: An ASP-based system for enterprise ontologies. *J. Log. Comput. 19*, 4 (2009), 643–670.

[110] Schrijvers, T., Stuckey, P. J., and Wadler, P. Monadic constraint programming. *J. Funct. Program. 19*, 6 (2009), 663–697.

[111] Shaw, M. Abstraction techniques in modern programming languages. *IEEE Software 1*, 4 (1984), 10–26.

[112] Shlyakhter, I., Seater, R., Jackson, D., Sridharan, M., and Taghdiri, M. Debugging overconstrained declarative models using unsatisfiable cores. In *ASE* (2003), IEEE Computer Society, pp. 94–105.

[113] Silva, J., Ed. *Functional and Constraint Logic Programming - 26th International Workshop, WFLP 2018, Frankfurt/Main, Germany, September 6, 2018, Revised Selected Papers* (2019), vol. 11285 of *Lecture Notes in Computer Science*, Springer.

[114] Simon Marlow, S. P.-J. The Glasgow Haskell Compiler, 2012.

[115] Snap. `http://snapframework.com/`. Accessed: 2015-07-01.

[116] socket.io. `http://socket.io/`. Accessed: 2015-07-01.

[117] Son, T. C., Pontelli, E., and Elkabani, I. An unfolding-based semantics for logic programming with aggregates. *CoRR abs/cs/0605038* (2006).

[118] Stockmeyer, L. J. The polynomial-time hierarchy. *Theor. Comput. Sci. 3*, 1 (1976), 1–22.

[119] Tasharrofi, S., and Ternovska, E. A semantic account for modularity in multi-language modelling of search problems. In *Frontiers of Combining Systems, 8th International Symposium, FroCoS 2011, Saarbrücken, Germany, October 5-7, 2011. Proceedings* (2011), C. Tinelli and V. Sofronie-Stokkermans, Eds., vol. 6989 of *Lecture Notes in Computer Science*, Springer, pp. 259–274.

[120] Tasharrofi, S., and Ternovska, E. Three semantics for modular systems. *CoRR abs/1405.1229* (2014).

[121] Taylor, J., Fish, A., Vanthienen, J., and Vincent, P. Emerging standards in decision modeling. In *Intelligent BPM Systems: Impact and Opportunity*. iBPMS Expo, 2013, pp. 133–146.

[122] AngularJS. https://angularjs.org/.

[123] van Fraassen, B. Singular terms, truth-value gaps and free logic. *Journal of Philosophy 63*, 17 (1966), 481–495.

[124] Van Gelder, A. The well-founded semantics of aggregation. In *PODS* (1992), ACM Press, pp. 127–138.

[125] Vanthienen, J. PROLOGA: from business knowledge modeling to ruleml. In *Rule Representation, Interchange and Reasoning on the Web. Series: LNCS, Vol. 5321 The International RuleML Symposium on Rule Interchange and Applications (RuleML-2008)* (2008), Springer, p. 3p.

[126] Vanthienen, J., and Caron, F. Modeling business decisions and processes–which comes first? In *Proceedings of the International Conference on Knowledge Discovery and Information Retrieval and the International Conference on Knowledge Management and Information Sharing* (2014), KMIS, pp. 451–456.

[127] Waddell, O., Sarkar, D., and Dybvig, R. K. Fixing letrec: A faithful yet efficient implementation of scheme's recursive binding construct. *Higher-Order and Symbolic Computation 18*, 3-4 (2005), 299–326.

[128] Wadler, P. Comprehending monads. *Mathematical Structures in Computer Science 2*, 4 (1992), 461–493.

[129] Wazny, J. R. *Type inference and type error diagnosis for Hindley/Milner with extensions*. Citeseer, 2006.

[130] Weber, I., Hoffmann, J., and Mendling, J. Beyond soundness: on the verification of semantic business process models. *Distributed and Parallel Databases 27*, 3 (2010), 271–343.

[131] Wittocx, J., Mariën, M., and Denecker, M. Grounding with bounds. In *AAAI* (2008), D. Fox and C. P. Gomes, Eds., AAAI Press, pp. 572–577.

[132] Wittocx, J., Mariën, M., and Denecker, M. Grounding FO and FO(ID) with bounds. *J. Artif. Intell. Res. (JAIR) 38* (2010), 223–269.

[133] Xtext: Language development made easy! https://www.eclipse.org/Xtext/, 2014.

List of Publications

A complete and up-to-date list of publications can be found at `www.cs.kuleuven.be/publicaties/lirias/mypubs.php?unum=U0093049`

Journal Articles

- I. Dasseville, M. van der Hallen, G. Janssens, M. Denecker: "Semantics of templates in a compositional framework for building logics." In: Theory and Practice of Logic Programming, volume 15, pages 681-695, (15 pages), 2015
- P. Van Hertum, I. Dasseville, G. Janssens, M. Denecker: "The KB Paradigm and its application to interactive configuration." In: Theory and Practice of Logic Programming, volume 17, pages 91-117, (27 pages), 2017

Peer-reviewed Articles at Conferences

- I. Dasseville, M. van der Hallen, B. Bogaerts, G. Janssens, M. Denecker. "A compositional typed higher-order logic with definitions." In: OpenAccess Series in Informatics, volume 52, page 14:1, Schloss Dagstuhl - Leibniz-Zentrum Fuer Informatik, International Conference On Logic Programming, New York City, Usa, 16/10/2016 - 21/10/2016
- P. Van Hertum, I. Dasseville, G. Janssens and M. Denecker. "The KB Paradigm and its application to interactive configuration." In: Marco Gavanelli and John H. Reppy (eds.), Practical Aspects of Declarative Languages - 18th International Symposium, (PADL) 2016, St. Petersburg, FL, USA, 18-19 January 2016. pages 13-29

- J. Jansen, I. Dasseville, J. Devriendt, G. Janssens. "Experimental evaluation of a state-of-the-art grounder". In: Principles and Practice of Declarative Programming, 8-10 September 2014, pages 249-259.

Peer-reviewed Articles at Workshops

- I. Dasseville, L. Janssens, G. Janssens, J. Vanthienen, M. Denecker. "Combining DMN and the knowledge base paradigm for flexible decision enactment." In: CEUR Workshop Proceedings, volume 1620, Ceur-Ws.org, Ruleml, Stony Brook, 6/7/2016 - 9/7/2016
- I. Dasseville, G. Janssens. "A web-based IDE for IDP" In: Proceedings of the International Workshop on User-Oriented Logic Programming (IULP 2015), pages 21-32
- I. Dasseville, G. Janssens. "FMS: Functional Programming as a Modelling Language" In: Functional and Constraint Logic Programming - 26th International Workshop, WFLP 2018, Revised Selected Papers (2019), pages 145-161, vol 11285 of Lecture Notes in Computer Science, Springer, edited by Silva J.
- I. Dasseville, M. Denecker. "Transpiling Programmable Computable Functions to Answer Set Programs" In: Functional and Constraint Logic Programming - 26th International Workshop, WFLP 2018, Revised Selected Papers (2019), pages 3-17, vol 11285 of Lecture Notes in Computer Science, Springer, edited by Silva J.
- M. Cramer, P. Van Hertum, R. Lapauw, I. Dasseville, M. Denecker. "Resilient Delegation Revocation with Precedence for Predecessors is NP-Complete" In: The Computer Security Foundations Workshop III, volume 2016-August, pages 432-442, (11 pages), IEEE, IEEE 29th Computer Security Foundations Symposium (Csf), Lisbon, Portugal, 27/6/2016 - 1/7/2016
- R. Lapauw, I. Dasseville, M. Denecker. "Visualising interactive inferences with IDPD3" In: Proceedings of the International Workshop on User-Oriented Logic Programming (IULP 2015), pages 67-81



FACULTY OF ENGINEERING SCIENCE
DEPARTMENT OF COMPUTER SCIENCE
DTAI
Celestijnenlaan 200A box 2402
B-3001 Leuven
krr@cs.kuleuven.be
http://www.dtai.cs.kuleuven.be