Compiling Constraint Handling Rules to Java: A Reconstruction

Peter Van Weert

Report CW 521, August 2008

Katholieke Universiteit Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

Compiling Constraint Handling Rules to Java: A Reconstruction

Peter Van Weert[*]

Report CW 521, August 2008

Department of Computer Science, K.U.Leuven

Abstract

In this report, we provide a detailed description of the compilation scheme the K.U.Leuven JCHR system uses to compile CHR to efficient Java code. We start from a relatively straightforward adaptation of the traditional CHR compilation scheme for Prolog, and gradually add all its basic optimizations. Next, we show why this compilation scheme is not suited for compilation to an imperative host language such as Java. We therefore introduce a novel compilation scheme from CHR to Java that uses explicit call stack maintenance and trampoline-style compilation to guarantee that executing recursive CHR programs no longer results in call stack overflows. The empirical evaluation of the improved compilation scheme confirms it is mostly superior to the traditional one.

Keywords : Constraint Handling Rules, compilation, optimization.
CR Subject Classification : D.3.2 [Programming Languages] Language Classifications — Constraint and logic languages, D.3.4 Processors — Code generation, Compilation, Optimization.

---

[*]Research Assistant of the Research Foundation– Flanders (FWO-Vlaanderen).

Contents

List of Code Listings

List of Figures

List of Tables

Chapter 1

Introduction

Constraint Handling Rules (CHR) [CHR08, Frü98, Frü08] is a high-level, elegant committed-choice CLP language, based on multi-headed, guarded multiset rewrite rules. Originally designed for the declarative specification of constraint solvers, CHR has matured toward a powerful general purpose language, used in a wide range of application domains, including computational linguistics, multi-agent systems, and type system design. A recent survey on CHR is [SVWSDK08].

Although CHR is Turing complete [SSD05b], it is typically implemented as a language extension, embedded in a host language. Traditional host languages for CHR are (constraint) logic programming languages, such as Prolog [HF00, SD04, Sch05a], HAL [DSGH03, HGSD05, Duc05], and Mercury. Efficient implementations also exist for other host languages, including Haskell [CSW03, SSW04], Java [Wol01, AKSS02, VSD05], and C [WSD07].

The first complete and efficient CHR system was [HF00]. This implementation has long been considered the *reference implementation* of CHR. In [Sch05a], a comprehensive explanation of its compilation schema is provided. In this report, we denote this scheme as the *traditional compilation scheme*. The operational semantics of [HF00] is captured formally by the *refined operational semantics* [DSGH04], which rapidly became the norm for later systems. The semantics and compilation scheme of [HF00] serve as the basis for most current, state-of-the-art CHR implementations, including the K.U.Leuven CHR system [SD04, Sch05a] for Prolog, and the systems for the imperative host languages Java [VSD05] and C [WSD07].

This work presents the compilation scheme used by the K.U.Leuven JCHR system [VSD05]. The K.U.Leuven JCHR system [VSD05, VW08a] is a state-of-the-art CHR implementation in Java. It features a statically typed declarative syntax, a tight integration with the object-oriented host-language, extensive static analysis, and a compilation to highly optimized code. For more information and examples on the use and syntax of the JCHR language, see [VW08a, VW08b].

The JCHR system and its compilation scheme influenced the design and implementation of the CCHR system [WSD07], the first CHR embedding in the C programming language. Thanks to its compilation to efficient low-level code, it can come close to native C code. Both our imperative CHR systems, JCHR and CCHR, outperform existing CHR implementations by up to several orders of magnitude.

A reworked version of this report appears in [VWWSD08]. It features a generalization of the compilation scheme presented here, and that of CCHR, towards any imperative target language. The article also outlines several other challenges faced when embedding CHR in an imperative host language, and contains a more detailed discussion of related work.

The presentation of the compilation in [VWWSD08] is more high-level, and contains a more complete survey of all compiler optimizations that appeared in recent literature (including [DS07, Duc05, DS05, HGSD05, Sch05a, SSD05a, SSD05c, SSD06b, VW08c]). This report on the other hand focusses on the compilation scheme of JCHR, and provides some more low level details and optimizations thereof. More advanced optimizations are not presented, though references to specialized literature are provided where relevant.

Overview The K.U.Leuven JCHR system implements the refined operational semantics [DSGH04]. Section 1.1 shortly reviews this semantics, as it is imperative for a good understanding of the compilation schemes presented in subsequent chapters.

Chapter 2 provides a reconstruction of the *traditional compilation scheme* used by JCHR for compiling CHR to Java. It is a relatively direct translation of the scheme used by Prolog embeddings. Sections 2.1 and 2.2 are a translated, extended revision of [VW05, Sections 8.2.4–8.2.5], and are analogous the corresponding chapter [Sch05b] in [Sch05a]. They provide a first thorough description of JCHR's traditional compilation scheme. The evaluation of this scheme in Section 2.3, however, reveals the traditional compilation scheme is less suited for compiling CHR to an imperative language such as Java. Similar results were observed for C [WSD07]. The main reason is the lack of tail call optimizations in imperative host languages. For recursive CHR programs, the traditional compilation scheme therefore rapidly results in call stack overflows.

Chapter 3 outlines a new and improved compilation scheme for CHR to Java that eliminates this issue completely. Evaluation in Section 3.3 shows the new compilation scheme is superior to the traditional one.

1.1 The Refined Operational Semantics ω_r

Basic knowledge of CHR, its syntax and its semantics, is assumed. Good introductions can be found for instance in [Duc05, Frü98, Frü08, Sch05a]. Only CHR's *refined operational semantics* is reviewed in more detail [DSGH04]. This semantics, commonly denoted as the ω_r (operational) semantics, formally captures the operational semantics of most current CHR implementations. The compilation schemes presented in Chapters 2 and 3 will be implementations of this operational semantics. This section revises the refined operational semantics. For more details, consult [DSGH04] or [Duc05].

The ω_r semantics is formulated as a state transition system. Transition rules define the relation between subsequent execution states in a CHR derivation. The version presented here follows [Duc05, Sch05a]. This slight modification of the original specification [DSGH04] describes more closely the semantics implemented by JCHR, and most other recent CHR implementations.

Notation Sets, multisets and sequences (ordered multisets) are defined as usual. We use $S[i]$ to denote the i'th element of a sequence S, $+\!\!+$ for sequence *concatenation*, and $[e|S]$ to denote $[e] +\!\!+ S$. The *disjoint union* of sets is defined as: $\forall X, Y, Z : X = Y \sqcup Z \leftrightarrow X = Y \cup Z \wedge Y \cap Z = \emptyset$. For a logical expression X and a set V of variables, $vars(X)$ denotes the set of *free variables*, and *constraint projection* is defined as $\pi_V(X) \leftrightarrow \exists v_1, \ldots, v_n : X$ with $\{v_1, \ldots, v_n\} = vars(X) \setminus V$.

Execution States An execution state of ω_r is a tuple $\langle \mathbb{A}, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n$. The role of the sequence $\mathbb{A}$, called the *execution stack*, is explained below, in the paragraph on ω_r's transition rules. The ω_r semantics is multiset-based. To distinguish between otherwise identical constraints, the *CHR constraint store* $\mathbb{S}$ is thus a set of *identified* CHR constraints, denoted $c\#i$, where each CHR constraint c is associated with a unique *constraint identifier* $i \in \mathbb{N}$. The projection operators $chr(c\#i) = c$ and $id(c\#i) = i$ are extended to sequences and sets in the obvious manner. The integer n represents the next integer to be used as a constraint identifier.

The *built-in constraint store* $\mathbb{B}$ is a conjunction containing all built-in constraints passed to the built-in solver. Their meaning is determined by the built-in constraint theory $\mathcal{D}_\mathcal{H}$ (see e.g. [Sch05a] for rigorous definition of this concept). The *propagation history* $\mathbb{T}$, finally, is a set of tuples, each recording a sequence of identifiers of CHR constraints that fired a rule, and the name of that rule.

Transition Rules Fig. 1.1 lists the transition rules of ω_r. Execution proceeds by exhaustively applying transitions, starting from an *initial execution state* of the form $\langle Q, \emptyset, \texttt{true}, \emptyset \rangle_1$. The constraint sequence Q is called the *query*.

1. Solve $\langle [b\vert\mathbb{A}], \mathbb{S}, \mathbb{B}, \mathbb{T}\rangle_n \rightarrowtail_{\mathcal{P}} \langle S \mathbin{++} \mathbb{A}, \mathbb{S}, b \wedge \mathbb{B}, \mathbb{T}\rangle_n$ if b is a built-in constraint. For the set of *reactivated constraints* $S \subseteq \mathbb{S}$, the following bounds hold: lower bound: $\forall H \subseteq \mathbb{S} : (\exists K, R : H = K \mathbin{++} R \wedge \exists \rho \in \mathcal{P} : \neg appl(\rho, K, R, \mathbb{B}) \wedge appl(\rho, K, R, b \wedge \mathbb{B})) \rightarrow (S \cap H \neq \emptyset)$ and upper bound: $\forall c \in S : vars(c) \not\subset fixed(\mathbb{B})$.
2. Activate $\langle [c\vert\mathbb{A}], \mathbb{S}, \mathbb{B}, \mathbb{T}\rangle_n \rightarrowtail_{\mathcal{P}} \langle [c\#n\!:\!1\vert\mathbb{A}], \{c\#n\} \sqcup \mathbb{S}, \mathbb{B}, \mathbb{T}\rangle_{n+1}$ if c is a CHR constraint (which has not yet been active or stored in $\mathbb{S}$).
3. Reactivate $\langle [c\#i\vert\mathbb{A}], \mathbb{S}, \mathbb{B}, \mathbb{T}\rangle_n \rightarrowtail_{\mathcal{P}} \langle [c\#i:1\vert\mathbb{A}], \mathbb{S}, \mathbb{B}, \mathbb{T}\rangle_n$ if c is a CHR constraint (re-added to $\mathbb{A}$ by a **Solve** transition but not yet active).
4. Simplify $\langle [c\#i\!:\!j\vert\mathbb{A}], \mathbb{S}, \mathbb{B}, \mathbb{T}\rangle_n \rightarrowtail_{\mathcal{P}} \langle B \mathbin{++} \mathbb{A}, K \sqcup S, \theta \wedge \mathbb{B}, \mathbb{T}'\rangle_n$ with $\mathbb{S} = \{c\#i\} \sqcup K \sqcup R_1 \sqcup R_2 \sqcup S$, if the j-th occurrence of c in $\mathcal{P}$ occurs in rule ρ, and θ is a matching substitution such that $apply(\rho, K, R_1 \mathbin{++} [c\#i] \mathbin{++} R_2, \mathbb{B}, \theta) = B$. Let $t = (\rho, id(K \mathbin{++} R_1) \mathbin{++} [i] \mathbin{++} id(R_2))$, then $t \notin \mathbb{T}$ and $\mathbb{T}' = \mathbb{T} \cup \{t\}$.
5. Propagate $\langle [c\#i\!:\!j\vert\mathbb{A}], \mathbb{S}, \mathbb{B}, \mathbb{T}\rangle_n \rightarrowtail_{\mathcal{P}} \langle B \mathbin{++} [c\#i\!:\!j\vert\mathbb{A}], \mathbb{S} \setminus R, \theta \wedge \mathbb{B}, \mathbb{T}'\rangle_n$ with $\mathbb{S} = \{c\#i\} \sqcup K_1 \sqcup K_2 \sqcup R \sqcup S$, if the j-th occurrence of c in $\mathcal{P}$ occurs in rule ρ, and θ is a matching substitution such that $apply(\rho, K_1 \mathbin{++} [c\#i] \mathbin{++} K_2, R, \mathbb{B}, \theta) = B$. Let $t = (\rho, id(K_1) \mathbin{++} [i] \mathbin{++} id(K_2 \mathbin{++} R))$, then $t \notin \mathbb{T}$ and $\mathbb{T}' = \mathbb{T} \cup \{t\}$.
6. Drop $\langle [c\#i\!:\!j\vert\mathbb{A}], \mathbb{S}, \mathbb{B}, \mathbb{T}\rangle_n \rightarrowtail_{\mathcal{P}} \langle \mathbb{A}, \mathbb{S}, \mathbb{B}, \mathbb{T}\rangle_n$ if c has no j-th occurrence in $\mathcal{P}$.
7. Default $\langle [c\#i\!:\!j\vert\mathbb{A}], \mathbb{S}, \mathbb{B}, \mathbb{T}\rangle_n \rightarrowtail_{\mathcal{P}} \langle [c\#i:j+1\vert\mathbb{A}], \mathbb{S}, \mathbb{B}, \mathbb{T}\rangle_n$ if the current state cannot fire any other transition.

Figure 1.1: The transition rules of the refined operational semantics ω_r.

The execution stack $\mathbb{A}$ is used to treat CHR constraints as function calls. The top-most element of $\mathbb{A}$ is called the *active constraint*. When active, a CHR constraint performs a search for partner constraints that match the head of a rule. The constraint's occurrences are tried in a top-down, right-to-left order. To realize this order in ω_r, identified constraints on the execution stack are *occurrenced* in **Activate** and **Reactivate** transitions. When an occurrenced identified CHR constraint $c\#i : j$ is active, only matches with the j'th occurrence of c's constraint type are considered. Interleaving a sequence of **Default** transitions, all applicable rules are fired in **Propagate** and **Simplify** transitions. A rule is applicable if the store contains matching partner constraints for all remaining occurrences in its head. Formally:

Definition 1.1 *Given a conjunction of built-in constraints $\mathbb{B}$, a rule ρ is* applicable *with sequences of identified CHR constraints K and R, denoted $appl(\rho, K, R, \mathbb{B})$, iff a matching substitution θ exists for which $apply(\rho, K, R, \mathbb{B}, \theta)$ is defined. The latter partial function is defined as $apply(\rho, K, R, \mathbb{B}, \theta) = B$ iff $K \cap R = \emptyset$ and, renamed apart, ρ is of form "ρ @ $H_k \setminus H_r \Leftrightarrow G\vert\ B$" ($H_k$ or H_r may be empty) with $chr(K) = \theta(H_k)$, $chr(R) = \theta(H_r)$, and $\mathcal{D}_{\mathcal{H}} \models \mathbb{B} \rightarrow \pi_{vars(\mathbb{B})}(\theta \wedge G)$.*

If the top-most element of $\mathbb{A}$ is a built-in constraint, this constraint is passed to the built-in solver in a **Solve** transition. As this may affect the entailment of guards, all CHR constraints for which additional rules might have become applicable have to be put back on the execution stack. These then cause **Reactivate** transitions to reinitiate searches for applicable rules. Constraints with fixed arguments are not reactivated, as no additional guards can become entailed.

Definition 1.2 *A variable v is* fixed *by constraint conjunction B, or $v \in fixed(B)$, iff $\mathcal{D}_{\mathcal{H}} \models \forall\theta((\pi_{\{v\}}(B) \wedge \pi_{\{\theta(v)\}}(\theta(B))) \rightarrow v = \theta(v))$ for any variable renaming θ.*

When a rule fires, its body is executed. By putting the body on the activation stack, the different conjuncts of the body are activated (for CHR constraints) or solved (for built-in constraints) in a left-to-right order. Control only returns to the original active constraint *after* the body is completely executed. This corresponds closely to the execution of procedure calls in the stack-based programming languages, such as Prolog and Java, to which CHR is compiled.

Chapter 2

Traditional Compilation Scheme

The compilation scheme commonly used to compile CHR to (constraint) logic programming languages is best described in [Sch05b]. This chapter is written to be analogous to the latter chapter of [Sch05a], and describes the result of porting this compilation scheme to the Java setting. Section 2.1 introduces a simplified scheme that closely follows the refined operational semantics (cf. Section 1.1). Next, in Section 2.2, several simple optimizations are added one by one. The more advanced analyses and optimizations performed by the system are outside the scope of this paper (Section 2.2 contains several footnote references that direct the interested reader to the relevant literature on the subject). Section 2.3, finally, evaluates this compilation scheme.

2.1 Basic Compilation Scheme

Where applicable, implementation aspects are related to the transition rules of the refined operational semantics, or the different components of the formal execution state. In short, an ω_r execution state $\langle \mathbb{A}, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle$ is implemented as follows:

- As in most traditional CHR compilation schemes, **Activate** and **Reactivate** transitions are implemented as procedure calls—here: Java method invocations. The *activation stack* $\mathbb{A}$ is thus mapped onto the implicit call stack of the Java Virtual Machine (JVM). In Chapter 3, a new compilation scheme is presented that maintains the activation stack more explicitly.
- $\mathbb{S}$, the CHR *constraint store*, is maintained by a generated `Handler` class, as described in Section 2.1.1. The concrete data structures used are beyond the scope of this report.
- In JCHR *built-in constraint solvers* are, as far as the JCHR compiler is concerned, black boxes. They are responsible for performing **Solve** transitions. The interaction with arbitrary built-in solvers is inspired by [DSGH03] (see also [VWWSD08]). For more details on JCHR's built-in constraint solvers, we refer to [VW08b].
- For the implementation of the propagation history $\mathbb{T}$, cf. Section 2.2. Details are again out of the scope of this text.

2.1.1 The `Handler` class

For each JCHR handler h, a single subclass of `Handler` is generated, named H`Handler`. For generic handlers the class is parameterized with the handler's type parameters (generic handlers are analogous to generic classes in Java: see [VW08b] for details).

The handler class contains references to all built-in constraint solvers declared in the handler source file. The main responsibility of a handler class though is the management of the *CHR constraint store*. The constraint store can be considered an abstract collection of `Constraint` objects[1]. A `Handler` class provides the following methods for each user-defined constraint $c(\overline{args})$ declared in its source file ($\overline{args}$ is a conjunction of arguments):

- `tell`$C(\overline{args})$: The `tell`$C(\overline{args})$ method for a constraint $c(\overline{args})$ is implemented as follows:

  ```
  public void tellC(args) {
      new CConstraint(args).activate();
  }
  ```

  The `activate()` method corresponds to an **Activate** transition in the refined operational semantics. Detailed information on the C`Constraint` classes is found in the Section 2.1.2.

- `store`C(C`Constraint`): stores the provided C`Constraint` object in the constraint store, updating all necessary data structures and indexes. Unlike the `public tell`C method, this method is only accessible from within the H`Handler` class.

- `lookup`$C()$: returns an `Iterator`<C`Constraint`> object [GHJV95] that allows traversal of all C`Constraint` objects in the constraint store[2]. The iterators have to obey the following three properties, which will be exploited (or relaxed) throughout Section 2.2:

  1. The returned iterator is robust under structural modifications of the underlying constraint store, that is: the iterator does not fail if C`Constraint`'s are added or removed during the iteration. Ideally, the next candidate partner constraint is returned in (amortized) $\mathcal{O}(1)$, even if arbitrary constraints may be removed during the iteration. Even for simple indexes, this can be quite challenging.
  2. Only live constraints are returned.
  3. A contiguous iteration does not contain duplicates. This is required for termination. Preferably an iteration remains duplicate-free under structural modifications of the constraint store.
  4. All constraints stored at the moment of the iterator's creation are returned at least once in the iteration. Constraints added after the creation of the iterator do not have to appear in the iteration. This is correct, since the refined operational semantics requires that these constraints were already activated earlier. All rules for which they appear in the matching have therefore already been fired, including those that include the current active constraint as well.

The implementation of the last two methods depends on the concrete data structures used for the constraint store, and on the indices used for efficient constraint retrieval. These concerns, however, are beyond the scope of this paper.

2.1.2 The `Constraint` classes

The reference SICStus Prolog implementation of CHR [HF00] reads:

> A CHR constraint is implemented as both *code* (a Prolog predicate)
> and *data* (a Prolog term in the constraint store).

The same principle is used here: a CHR constraint is implemented as a Java object serving as *data* in the constraint store, whose class contains the *code* that has to be executed when the constraint is active (cf. the refined operational semantics).

[1] Each `Handler` class effectively implements the standard `java.util.Collections<Constraint>` interface.

[2] The actual implementation will offer a series of different `lookup`$C(\ldots)$ methods. using different constraint store indexes for more efficient constraint retrieval. Details are beyond the scope of the paper.

⟨*arguments*⟩	✓	For each formal argument of the constraint, an instance field is generated with the same type as declared. The name of a formal argument, and thus the corresponding field, can also be declared to improve usability.
`int ID`		A unique increasing number assigned on constraint object creation. This field corresponds to the constraint identifier used by the refined operational semantics. It is used for propagation histories, and for sorting lists in certain constraint store implementations.
`boolean alive`		Indicates whether a constraint is *alive* or not. A constraint *dies* when its `terminate()` method is called. In the refined operational semantics this corresponds more or less to the removal of the constraint from the constraint store in a **Simplify** transition. In practice though, the constraint is not necessarily stored before it is terminated (cf. the *Late Storage Optimization* in Section 2.2).
`boolean stored`		Indicates whether a constraint is stored in the constraint store. Unlike in the refined operational semantics, the optimized compilation scheme only effectively stores constraints if necessary (cf. the *Late Storage Optimization* in Section 2.2).
⟨*history*⟩	✓	A constraint can contain multiple instance fields related to the propagation history (one for each propagation rule): for details see *Distributed Propagation History* in Section 2.2.
`boolean reactivated`	✓	Used for the *Drop after Reactivation* optimization in Section 2.2.
⟨*handler*⟩	✓	Each constraint has a reference to the constraint handler that manages it. As `Constraint` classes are generated as non-static inner classes of a handler, this reference is implicit. This way, `Constraint` classes gain access to the methods and fields of their enclosing `Handler` class.

Table 2.1: Overview of the instance fields of a constraint. Generated fields are indicated with ✓; the other fields are inherited from the abstract super class `Constraint`.

2.1.2.1 JCHR `Constraints` as Data

For each user-defined constraint c declared in a handler, a class C`Constraint` is generated, extending the abstract `Constraint` class. These constraint classes are inner classes of their handler class, thus implicitly obtaining access to the built-in solver references and all constraint store methods. For parameterized handlers, the handler type parameters can also be used inside the constraint classes. Table 2.1 provides an overview of the instance fields of the generated constraint classes. Public inspector methods are inherited or generated where applicable.

2.1.2.2 JCHR `Constraints` as Code

Listing 2.1 shows the first part of the compilation scheme for a constraint c with n occurrences. The `activate()` method corresponds to the **Activate** transition of the refined operational semantics, and is called immediately after creation of the constraint object. First, as in ω_r, the constraint is stored in the constraint store. The rule preceded with `#` is pseudo-code: the constraint registers itself as an *observer* with the different non-fixed arguments of the constraint. The `notify()` method of the well-known *observer pattern* [GHJV95] is called `reactivate()`, and corresponds to a **Reactivate** transition. This method is called by a built-in constraint solver in what corresponds to a **Solve** transition: each time a new built-in constraint has been told, all JCHR constraints

```
public void activate() {
    store();
    occurrences();
}

protected void store() {
    storeC(this);
#   foreach (a : non-fixed arguments of c)
        a.addBuiltInConstraintObserver(this);
}

public void reactivate() {
    occurrences();
}

private final void occurrences() {
    c_1(); c_2(); ... c_n();
}
```

Listing 2.1: Basic compilation scheme for (re)activations of a constraint c with n occurrences.

observing the variables involved are reactivated.

In ω_r, the **Activate** and **Reactivate** transitions put an identified constraint on the execution stack. The basic compilation scheme implements this implicitly: each constraint receives a unique identifier upon creation (cf. Table 2.1), and by calling the `activate()` or `reactivate()` method, a stack frame is added to the Java Virtual Machine call stack.

An **Activate** or **Reactivate** transition is always followed by a sequence of **Default** or **Propagate** transitions, and then a **Simplify** or **Drop** transition. This is realized in the `occurrences()` method called by both `activate()` and `reactivate()`. This method then calls a so-called *occurrence method* for each of the n occurrences of the constraint c. The order in which the occurrences are traversed is the one determined by the refined operational semantics.

The remainder of this section presents the compilation scheme for the occurrence methods of a rule of the following generic form:

$$\begin{array}{l}\rho \,@\, \mathrm{c}_1^{[j_1]}(\mathrm{X}_{1,1},\,\ldots,\,\mathrm{X}_{1,a_1}),\,\ldots,\,\mathrm{c}_{r-1}^{[j_{r-1}]}(\mathrm{X}_{r-1,1},\,\ldots,\,\mathrm{X}_{r-1,a_{r-1}}) \setminus \\ \qquad \mathrm{c}_r^{[j_r]}(\mathrm{X}_{r,1},\,\ldots,\,\mathrm{X}_{r,a_r}),\,\ldots,\,\mathrm{c}_h^{[j_h]}(\mathrm{X}_{h,1},\,\ldots,\,\mathrm{X}_{h,a_h}) \Leftrightarrow \mathrm{g}_1,\,\ldots,\,\mathrm{g}_{n_g} \mid \mathrm{b}_1,\,\ldots,\,\mathrm{b}_{n_b}.\end{array}$$

Here, a rule ρ has h occurrences $\mathrm{c}_i^{[j_i]}$ in its head, numbered from left to right, with r the index of the first removed occurrence. For a simplification rule there are no kept occurrences (i.e., $r = 1$), for a propagation rule there are no removed occurrences ($h = r-1$). The occurrence number j_i of an occurrence $\mathrm{c}_i^{[j_i]}$ denotes that this occurrence is the j_i'th occurrence of constraint c_i in the program, when numbered according to the top-to-bottom, right-to-left order determined by ω_r.

Listing 2.2 lists the basic compilation scheme for an occurrence $\mathrm{c}_i^{[j_i]}$ of such a rule. The following pseudo-code operators are used:

- `#type(`x`)` returns the Java type of its argument x; and
- `#arg(`c`,`i`)` returns the the i'th formal argument of the constraint c.

Lines 7–30 implements the search for partner constraints, in the order they appear left-to-right in the head of the rule[3]. Partner constraints are retrieved using the `lookupC` methods offered by

[3] In the actual implementation the order in which partner constraints are searched is determined by means of static analysis. The problem of determining this optimal join order is addressed comprehensively in [DS07].

```
private final void c_i_j_i() {
   final C_iConstraint c_i_j_i = this;
   final #type(#arg(c_i,1)) X_{i,1} = this.#arg(c_i,1);
   ...
   final #type(#arg(c_i,a_i)) X_{i,m_i} = this.#arg(c_i,a_i);

   final Iterator<C_1Constraint> c_1_j_1_iter = lookupC_1();
   C_1Constraint c_1_j_1;
   #type(#arg(c_1,1)) X_{1,1};
   ...
   #type(#arg(c_1,a_1)) X_{1,m_1};

   while (c_1_j_1_iter.hasNext()) {
      c_1_j_1 = c_1_j_1_iter.next();
      X_{1,1} = c_1_j_1.#arg(c_1,1);
      ...
      X_{1,m_1} = c_1_j_1.#arg(c_1,a_1);

      final Iterator<C_2Constraint)> c_2_j_2_iter = lookupC_2();
      C_2Constraint c_2_j_2;
      #type(#arg(c_2,1)) X_{2,1};
      ...
      #type(#arg(c_2,a_2)) X_{2,m_2};

      while (c_2_j_2_iter.hasNext()) {
         c_2_j_2 = c_1_j_1_iter.next();
         X_{2,1} = c_2_j_2.#arg(c_2,1);
         ...
         X_{2,m_2} = c_2_j_2.#arg(c_2,a_2);
         ⋱
            if (c_1_j_1.isAlive()) {
               ⋱
                  if (c_h_j_h.isAlive()) {
                     if (c_1_j_1 != c_2_j_2 && ... && c_{h-1}_j_{h-1} != c_h_j_h) {
                        if (g_1) {
                           ⋱
                              if (g_{n_g}) {
                                 if (!inHistory_ρ(c_1_j_1, ..., c_n_j_n) {
                                    addToHistory_ρ(c_1_j_1, ..., c_n_j_n);
                                    c_r_j_r.terminate();
                                    ⋮
                                    c_h_j_h.terminate();

                                    b_1; ...; b_{n_b};
                                 }
                              }
                           ⋰
                        }
                     }
                  }
               ⋰
            }
         ⋰
      }
   }
}
```

Listing 2.2: Basic compilation for an occurrence.

the enclosing `Handler` class[4], and the *iterator pattern* [GHJV95] is used to iterate over all possible partners. For the active occurrence $c_i^{[j_i]}$ itself (lines 1–5), no iterator or `while` loop is generated. Lines 7–30 therefore consists of $n-1$ nested `while` loops.

Eventually, for each possible combination of partner constraints found by these nested loops, lines 31–38 check whether:

- all partner constraints are still alive (lines 31–33),
- all partner constraints are mutually distinct (line 34),
- all guards are satisfied (lines 35–37)[5],
- and whether that particular combination of constraints has not yet fired rule ρ, i.e., whether the combination is not yet present in the propagation history (line 38).

If all above tests succeed, the rule *fires*: the propagation history is extended (line 39), the constraints matching the removed occurrences are terminated (lines 40–42), and the body is executed (line 43). For $i < r$ this corresponds to a **Propagate** transition, for $i \geq r$ to a **Simplify** transition.

Guard and body conjuncts Listing 2.2 makes abstraction of the conjuncts of the guard and the body. As in the refined semantics, body conjuncts are executed left-to-right. A body conjunct `c(...)` with `c` a JCHR constraint is compiled as:

```
new CConstraint(...).activate();
```

The arguments of such a constraint are generally copied as is from the source code, possibly after some coercion and desugaring. The same applies for other conjuncts, that is built-in constraints and host language statements. More details can be found in [VW08b].

2.2 Standard Optimizations

This section lists several simple optimizations to the basic compilation scheme. This section again follows [Sch05b]: all "simple optimizations" listed [Sch05a, Section 5.3] are ported to the Java setting and listed here. Some extra optimizations, such as for instance *Backjumping*, are introduced as well.

The optimizations are grouped in three categories: *semantical* optimizations, *host language* optimizations, and *data structure* optimizations. [$\mathcal{S}$]emantical optimizations are based on reasoning about the (refined) operational semantics of CHR. [$\mathcal{H}$]ost language optimizations are optimizations tailored to generate more efficient host language code, adapted here in particular to the host language Java. [$\mathcal{D}$]ata structure optimizations finally deploy optimized data structures.

Early Distinct Partner Constraints Testing [$\mathcal{S}$]

Line 34 in Listing 2.2, added to remain analogous to [Sch05b], does not reflect what is done in the actual compilation scheme: each time a candidate partner constraint is found (i.e., after lines 14, 26, ...), this constraint is *immediately* compared to previous partners. Clearly, searching for more partners is useless, if the partners already found are not distinct. Constraints are also *only* compared to partners of the *same constraint type*.

[4] Commonly a more specific `lookupC` method will be used which takes advantage of maintained constraint store indexes to optimize the retrieval of partner constraint candidates.

[5] In the actual compilation scheme guards are tested as early as possible, i.e., as soon as all involved variables are bound by a candidate partner constraint. This is an instance of a standard compiler optimization technique called *loop-invariant code motion* (also: *hoisting*, or *scalar promotion*).

Propagation History Maintenance [$\mathcal{S}$]

The propagation is added to the CHR semantics to prevent the same rule to be applied multiple times with the same combination of constraints, thus avoiding trivial non-termination. Terminating one or more partner constraint already ensures the rule is fired only once with a particular combination. This means (as the observant reader already guessed from its name) that a *propagation* history only has to be maintained for *propagation* rules[6].

Distributed Propagation History [$\mathcal{D}$]

In JCHR, the propagation history is not maintained globally, but stored distributively in the different constraint objects (cf. Table 2.1). For each **Propagate** transition a new tuple is added to the part of the propagation history maintained by the active constraint. Consequently, each time the propagation history has to be checked for the presence of some tuple, only the parts of all involved partner constraints have to be checked.

Whereas both the SICStus reference implementation and the K.U.Leuven CHR system use AVL trees, the K.U.Leuven JCHR system uses data structures based on efficient hashing techniques to maintain the distributed parts of the propagation history. This reduces the time complexity of both checking and extending the propagation history from $\mathcal{O}(m\ log(n))$ to $\mathcal{O}(m)$, with m the number of occurrences in the head (typically only a few), and n the number of constraints in the constraint store (can become very large).

The propagation histories are also further specialized for each separate rule (unlike in the refined operational semantics, the rule identifier is therefore not explicitly part of the tuple stored in the data structure). This allows a further optimization: for propagation rules with only a single occurrence in the head, the propagation history is simply a `boolean`, maintained in the active constraint. For two-headed rules, more efficient data structures are possible as well: see [VW08d].

Even though the *time complexity* of a centralized propagation history would remain similar, the distributed approach is more favorable for memory reuse: if a constraint is terminated, its instance fields belonging to the distributed propagation history are set to `null`, allowing the JVM garbage collector to reclaim that part of the memory. A global propagation history on the other hand would maintain the reachability of all history tuples, unless custom memory management releases the memory explicitly. Whilst the risk for excessive, redundant memory use is not completely eliminated with the distributed propagation history maintenance, practice shows that it performs adequately. For more information on propagation history implementation, we refer to [VW08d].

Simplify Transitions [$\mathcal{S}$]

In a **Simplify** transition, the active constraint is removed. So, afterwards, it is clearly no longer necessary to keep looking for partner constraints. Consequently, in Listing 2.2 a "`return;`" statement can be added after line 44.

A second observation in the case of a **Simplify** transition is that the partner constraints are always alive if lines 31–33 of Listing 2.2 are reached. They were live when returned by their iterator (cf. Section 2.1.1), and could only become terminated when the rule is applied. By the optimization discussed in the previous paragraph however, the search for partner constraints is stopped after the first application of a **Simplify** transition. Lines 31–33 can thus safely be removed. Note that "`if (!alive) return;`" would have to be added prior to line 2, as otherwise rules could be fired with a active constraint that is terminated by an earlier occurrence method. The next optimization though will ensure this is no longer necessary.

Sequential Control Flow [$\mathcal{H}$]

Listing 2.2 uses a *sequential* approach: all different occurrence methods are called in sequence. Both the SICStus reference implementation and the K.U.Leuven CHR system however use a

[6] In fact, even for most propagation rules no history has to be maintained. We showed this in [VW08d] for non-reactive propagation rules, and, more recently, in [VW08c] for idempotent propagation rules.

```
            ⋮
            c_r_j_r.terminate();
            ⋮
            c_h_j_h.terminate();

            b_1; ...; b_n_b;

            return false;
        }
      ⋰
    }
    return true;
}
```

Listing 2.3: Optimized compilation scheme for a **Simplify** transition (using sequential control flow). Note that it is considerably less complicated than the compilation scheme for a **Propagate** transition (Listing 2.6).

continuation based approach: an occurrence predicate calls the next occurrence predicate if the active constraint is still alive (a *live continuation*), or does nothing if the latter is terminated (a *dead continuation*). This approach is also referred to as *chaining*.

In the case of an imperative language such as Java, chaining is not advised. The reason is that, contrary to Prolog, Java does not perform particularly well with deeply nested method calls. Therefore, a more sequential approach is preferred. There is nevertheless no point in calling occurrence methods once the active constraint has been terminated. To differentiate between life and dead continuations, the occurrence methods are modified to return a boolean: `true` in the case of a live continuation, `false` in the case of a dead continuation. The `occurrences()` and method from Listing 2.2 is then replaced with:

```
private final void occurrences() {
    if (c_1() && c_2() && ... && c_n());
}
```

Here the *laziness* of Java's '`&&`'-operator is exploited to ensure, essentially, that a **Drop** transition occurs as soon as the active constraint is terminated. This approach ensures the active constraint is always alive whilst searching for partner constraints. This also means that this no longer has to be tested in lines 31–33 of Listing 2.2. Recall that the *Simplification Transition* optimization already allowed the removal of all these lines in case of a **Simplify** transition.

The final, optimized compilation scheme for **Simplify** transitions is found in Listing 2.3. The following changes were performed to the basic compilation scheme of Listing 2.2 (this also includes the *Simplification Transition* optimization):

- Add "`return false;`" after line 44.
- Remove lines 31–33 entirely.
- Add as the last statement of the method (i.e., after line 55), "`return true;`", unless the **Simplify** transition is known to *always* be applied. The latter is the case if it is a single-headed simplification rule without guard.

In case of a **Propagate** transition, the following refinements are made to the basic compilation scheme:

```
1           ⋮
2           c_r_j_r.terminate();
3           ⋮
4           c_h_j_h.terminate();
5
6           b_1; ...; b_nb;
7
8           if (! alive) return false;
9
10          if (!c_1_j_1.isAlive()) continue label_c_1_j_1;
11          ⋮
12          if (!c_r-1_j_r-1.isAlive()) continue label_c_r-1_j_r-1;
13
14          continue label_c_r_j_r;
15      }
16  ⋰
```

Listing 2.4: Adding backjumping to the compilation scheme: replace lines 40–45 of 2.2 with the above code in case of a **Propagate** transition (also using sequential control flow). Each `while` loop is furthermore annotated with a label: `label_c`$_i$`_j`$_i$ for the loop iterating over the candidates for the i'th partner `c`$_i$`_j`$_i$.

- Even though the transition does not terminate the active constraint directly, it is possible the active constraint is terminated indirectly by executing the body of the rule. Therefore, "`if (!alive) return false;`" is added after line 44 (unless the body is empty).
- Remove the "`c`$_i$`_j`$_i$`.isAlive()`" test from lines 31–33.
- Add "`return true;`" as the last statement of the method.

The next three subsections further optimize the **Propagate** case.

Backjumping [$\mathcal{H}$]

Prolog implementations often use *backtracking* to search for matching partner constraints. The compilation scheme presented in Listing 2.2 on the other hand uses a pure iterative version, consisting of nested loops. This can be exploited in the case of **Propagate** transitions.

As seen in Section 2.1.1, constraint iterators only return live constraints. This means that the *first time* lines 31–33 are reached, these tests are superfluous (i.e., not only for the active constraint as already indicated in the previous subsection). After the application of a **Propagate** transition, generally more constraint combinations have to be searched (unless the active constraint is removed indirectly by the application of the body). If the corresponding rule is a simpagation rule though, certain partner constraints were terminated *directly*. Also, other partners may be removed *indirectly* by the execution of the body. Lines 31–33 therefore cannot safely be removed, as was the case with a **Simplify** transition.

However, relying on the refined operational semantics, the iterative control flow can be exploited to obtain a form of *backjumping*. Concretely, lines 31–33 are removed, and lines 40–45 are replaced with the code listed in Listing 2.4. To enable backjumping (lines 10–14), each `while` loop is also annotated with a *label*: `label_c`$_i$`_j`$_i$ for the loop that corresponds to occurrence $c_i^{[j_i]}$. This allows for instance the control to immediately jump to the outer most `while` loop if the first partner constraint no longer is alive. Without this optimization, a phenomenon referred to as *trashing* would occur: first all inner loops are iterated until exhaustion, each time failing because the first partner constraint is already dead, prior to eventually advancing the outer iterator. The backtracking used by Prolog implementations also suffers from trashing.

Listing 2.4 may warrant some more clarification. The constraints terminated directly by the rule application are never tested for their liveness (cf. line 14), saving some more tests compared to the basic compilation scheme. Furthermore in lines 10–12 the liveness of the active constraint is not tested again (cf. line 8). A final optimization is applicable if the body is empty: in that case, lines 8–12 are simply removed.

Existential Constraint Iterators $[\mathcal{D}, \mathcal{S}]$

In Section 2.1.1, the general requirements for constraint iterators were established. Several of these requirements are related to the behavior of the iterators under structural modifications on the constraint store. We denote such iterators *universal constraint iterators*. Implementing such iterators efficiently, however, is challenging. We therefore introduced a second type of constraint iterators, called *existential constraint iterators*, for which correct behavior is only guaranteed if during their lifetime no constraints are added or removed from the underlying constraint store. These iterators can often be implemented more efficiently (empirical results show up to 15% faster constraint traversal for linked lists).

Clearly, all iterators for `Simplify` transitions can safely be replaced with existential iterators. A similar optimization is also presented in [Sch05a]. The *Explicit Backjumping* optimization, however, allows existential iterators to be used in the compilation scheme of `Propagate` transitions as well: because control always jumps back at least as high as the loop for the first removed partner, existential iterators may be used for all removed partner occurrences. This optimization was inspired by [Wui07].

Drop after Reactivation $[\mathcal{S}]$

Executing the body of a rule after a **Propagate** transition may cause the active constraint to be reactivated (due to a **Solve** transition). In terms of the refined operational semantics, this means the same (identified) constraint occurs more than once on the activation stack. Based on this semantics, [Sch05a] proves that in that case all occurrences except the most upper one can safely be removed from the stack.

Traditionally, this is implemented using an integer field in the constraint representations, incremented each time a constraint is reactivated. In [Sch05b], this optimization is called *Generations*. Here, we introduce a more efficient implementation, that also transfers better to the compilation scheme of Chapter 3.

We use the `reactivated` boolean field in the `Constraint` classes (see Table 2.1). Before the body is executed, that is, at line 5 of Listing 2.4, the following line of code is added:

```
reactivated = false;
```

After each reactivation, the `reactivated` field is set to `true`: see Listing 2.5. Note that it is imperative that the field is only updated *after* the reactivation, otherwise nested reactivations would not work correctly. After the body is executed the field is tested. If the active constraint was reactivated, a **Drop** transition occurs. Concretely, in Listing 2.4, the following code is added after line 6:

```
if (reactivated) return false;
```

This optimization is of course not applied if the body is empty. The optimization is also pointless if none of the occurrences in the head will ever be reactivated, for instance if all arguments in the head are fixed. In all other cases, this optimization may save a lot of redundant work.

Late Storage $[\mathcal{S}]$

The refined operational semantics suggests a CHR constraint is added to the constraint store immediately at an **Activate** transition. This is also reflected in Listing 2.2. Often, however, a

```
public void activate() {
    if (occurrences()) store();
}

protected void store() {
    if (! stored) {
        stored = true;
        storeC(this);
#       foreach (a : non-fixed arguments of C)
            a.addBuiltInConstraintObserver(this);
    }
}

public void reactivate() {
    occurrences();
    reactivated = true;
}

private final boolean occurrences() {
    return c_1() && c_2() && ... && c_n();
}
```

Listing 2.5: Optimized compilation scheme for (re)activations of a constraint c with n occurrences (after adding the *Drop after Reactivation* and *Late Storage* optimizations).

constraint's lifetime is very short. The most obvious case is when an active constraint is terminated in a **Simplify** transition, shortly after activation. The constraint may also be terminated early indirectly, due to the execution of a body.

The goal of the *late storage* optimization is to postpone the addition of the constraint to the constraint store as long as possible. Consequently, in many cases, the constraint is effectively never stored at all, thus avoiding the considerable overhead of adding and removing the constraint to the constraint store. This subsection only describes the simple late storage optimization, used also by the reference SICStus Prolog implementation. More advanced instances use results of the so-called observation analysis [Duc05, SSD05a, Sch05a] to further delay constraint storage.

Execution points where a constraint has to be stored are at **Drop** transitions, and prior to the execution of a rule's body in **Propagate** transitions. The former is realized by moving the call to `store()` after all occurrences are tried: only if the last continuation is a live continuation, the constraint will be stored. The resulting compilation scheme is depicted in Listing 2.5. The `stored` field (cf. Table 2.1) is used to prevent the constraint from being added more than once. In the `terminate()` method it is also used to test whether a constraint has to be removed from the constraint store or not.

For **Propagate** transitions, the final, optimized compilation scheme is listed in Listing 2.6. It replaces the scheme of Listing 2.4. Lines 6–20 of Listing 2.6 are only generated if the body of the rule is non-empty[7].

Inlining [$\mathcal{H}$]

A final optimization discussed in [Sch05a] is *inlining*. There, as explained before, an occurrence predicates calls the next occurrence predicate in case of a live continuation, thus forming a chain of predicate calls (hence the term: *chaining*). Instead of calling the predicate, the predicate is often replaced by its body, which saves the overhead of doing the actual call.

[7] The compiler actually uses results from static analyses to determine whether a constraint can be added or reactivated by a rule's body. This analysis, called *observation analysis*, is described in detail in [Sch05a, SSD05a].

```
            ⋮
            c_{r}_j_{r}.terminate();
            ⋮
            c_{h}_j_{h}.terminate();

            reactivated = false;

            store();

            b_{1}; ...; b_{n_b};

            if (reactivated) return false;

            if (! alive) return false;

            if (!c_{1}_j_{1}.isAlive()) continue label_c_{1}_j_{1};
            ⋮
            if (!c_{r-1}_j_{r-1}.isAlive()) continue label_c_{r-1}_j_{r-1};

            continue label_c_{r}_j_{r};
        }
      ⋰
    }
    return true;
}
```

Listing 2.6: Optimized compilation scheme for a **Propagate** transition (using sequential continuation flow, and after adding the *Drop after Reactivation*, *Late Storage* and *Explicit Backjumping* optimizations).

It would be possible to inline the different occurrence methods into the `occurrences()` method of Listing 2.5. This is known to improve performance, as it may save many method calls. There are some reasons not to perform this form of inlining:

- Arguably, it decreases the readability of the generated code.
- In certain cases it is interesting to treat the `activate()` and `reactivate` method separately, as in general not all occurrences have to be retried in case of a **Reactivate** transition (cf. [DSGH03]). Inlining the occurrences in both the `activate()` and `reactivate()` method would duplicate code.
- The amount of code per method is limited to 65536 bytes in Java [LY99].

Currently, occurrence methods are not inlined, even though it would be interesting to investigate this possibility further in the future. Inlining is already applied though by the compiler to many auxiliary methods, including e.g. the `store()` and `occurrences` methods. For readability, the code samples in this document are listed without inlining.

2.3 Evaluation

For a performance comparison of JCHR with two other Java embeddings of CHR, DJCHR [Wol01] and JaCK [AKSS02], we refer to [VSD05]. In this work it was shown that JCHR, using the traditional compilation scheme presented in this chapter, outperforms these systems by up to several orders of magnitude. Because neither of these systems has evolved since, and JCHR's performance has only improved, this result remain valid.

In [VSD05], JCHR is also compared against two Prolog implementations of CHR. All these systems have advanced considerably though over the past years, An updated performance comparison is given at the end of next chapter (Section 3.3). In this chapter a new, improved compilation scheme for JCHR is presented, which solves the issues discussed in the following subsection.

2.3.1 Call stack overflows

The CHR language does not provide language primitives for loops. Other programming languages that share this property include most functional and logic programming languages, as well as some object-oriented languages such as SmallTalk. Any non-trivial CHR program therefore contains *recursion*. That is: directly or indirectly, there are rules with an occurrence of constraint c/n in the head that activate a body that adds c/n constraint to the store. While recursion has the same expressive power as iteration, recursive calls risk consuming a lot of stack space.

For recursive CHR programs, the traditional compilation schema generates a set of mutually recursive host language procedures. Unless the host language compiler or interpreter adequately deals with recursion, the traditional, call-based compilation scheme leads to stack overflow issues, as shown empirically below.

Languages that advocate a loop-free programming style employ several optimizations to execute recursion more efficiently, preferably within constant stack space. Prolog implementations e.g. perform *tail call optimization* since the early days of Prolog [War80]. This optimization consists in reusing the execution frame of the caller for the last call in the body of a clause. In other words: tail calls are executed by Prolog in constant stack space.

In CHR, a *tail call* occurs when the active constraint matches a removed occurrence, and the body ends with the addition of a CHR constraint. If the active constraint is not removed, the last body conjunct is not a tail call, as the search for partner constraints has to be resumed after the execution for the body, or more occurrences have to be tried for the previously active constraint.

For a host language such as Prolog, the traditional compilation scheme is therefore less problematic; indeed: to solve stack overflows during a CHR program's execution, it mostly suffices to rewrite the program to use tail calls for the recursive constraints. The underlying compiler or interpreter should then execute the program in constant stack space.

	JCHR		CCHR	SWI	YAP
	JRE 1.5	JRE 1.6			
tail	35,900	3,200	∞	∞	∞
non-tail	38,700	3,200	0.5M	3.3M	$\pm\infty$

Table 2.2: Recursion limits for different CHR systems. The number indicate the approximate value of N for which the handler below resulted in stack overflow for resp. JCHR and SWI when called with initial query `stack(N)`; ∞ indicates the program ran in constant space, and $\pm\infty$ indicates the only limit was available (virtual) memory.

Even though similar tail call optimizations are possible in imperative host languages (see e.g. [Pro01]), in practice, most compilers for imperative languages do not perform them, or only in certain situations. The GCC C compiler [Fre08], for instance, only optimizes tail calls in specific cases [Bau03]. Implementations of the Java Virtual Machine [LY99], including Sun's reference implementation HotSpot [Sun08], typically do not perform tail call optimizations at all[8]. Indeed, in practice, we observe that the traditional compilation schema for Java overflows the execution stack very quickly. For C the situation is only slightly better.

Empirical Verification

To test the limits on recursion we used the following simple CHR handler:

```
stack(0) <=> true.
stack(X) <=> stack(X-1).
```

We compared JCHR with CCHR [WSD07], and the K.U.Leuven CHR system [SD04, Sch05a] implementation for SWI-Prolog [Wie03, SWD05] and YAP Prolog [SC$^+$]. YAP Prolog is a more efficient Prolog system, but the YAP port uses an older version of the K.U.Leuven CHR system. The results[9] are given in Table 2.2.

Clearly, the second rule of the above handler contains a tail call. For all systems but JCHR, including CCHR, the host language compiler or runtime was able to perform the required tail call optimizations to run the program in constant space. Executing the compiled JCHR handler with the HotSpot Client JVM [Sun08], however, rapidly resulted in stack overflow. For the JRE 1.5 version stack overflow occurred for `N` equal to 35,900, for JRE 1.6 the situation even worsened.

We then altered the second rule to use non-tail recursion by adding an instruction after the recursive call. The results are shown in the second row of Table 2.2. The results for JCHR remained unchanged. For both SWI Prolog and CCHR, the native call stack has a static upper bound. For CCHR, the test resulted in stack overflow after around half a million recursive calls, for SWI after around 3.3 million. YAP Prolog's call stack grows dynamically, so YAP is only limited by available (virtual) memory.

We also tested the limits for different more realistic CHR benchmark programs. The results[10] can be found in Table 2.3. The numbers confirm the traditional compilation scheme is ill-suited for compiling CHR to Java. Due to the lack of tail call optimizations and the limited size of the call stack, stack overflows occur unacceptably fast when executing recursive JCHR programs. Depending on the version and platform, this can already be after a few thousand recursive calls. As most CHR programs contain some form of recursion, this problem is particularly severe. The next chapter proposes a new compilation scheme that completely solves these stack overflow issues.

[8] Java folklore suggests that supporting tail call optimization would interfere with Java's stack walking security mechanism (though this has recently been challenged in [CF04]).

[9] The tests of Table 2.2 were performed on a Intel® Core™ 2 Duo 6400 system with 2 GiB of RAM. SWI-Prolog 5.6.50 and YAP 5.1.2 were used. All C programs were compiled with GCC 4.1.3. K.U.Leuven JCHR 1.6.0 was used, and the generated Java code was compiled with Sun's JDK and executed with HotSpot JRE.

[10] The benchmarks of Table 2.3 were performed on a Intel® Pentium® 4 CPU 2.80GHz with 1GiB of RAM. K.U.Leuven JCHR 1.6.1 and SWI-Prolog 5.6.55 were used. The Java code was compiled with Sun's JDK 1.6.0 and executed with HotSpot JRE 1.6.0.

	JCHR	SWI	Description
`beer`(N)	3,500	∞	Sing the well-known 'N Bottles of Beer' song. The original program by Jon Sneyers was taken from `http://99-bottles-of-beer.net/`.
`dijkstra`(N)	2,100	∞	Using Dijkstra's algorithm to find the shortest path in a sparse graph with 16,384 nodes and 65,536 edges. A Fibonacci heap, also implemented in CHR, is used to obtain the optimal complexity (see [SSD06a] for a description of the Dijkstra and Fibonacci heap handlers).
`fibbo`(N)	1,800	*timeout*	Bottom-up computation of the N first Fibonacci numbers (origin: [Frü05]; see also [VW08c]).
`gcd`(N)	4,300	4.5M	Compute the greatest common divisor of N and 2 using Euclid's algorithm (classical example found on [CHR08]).
`primes`(N)	4,800	∞	Determine all primes numbers up to N using the Sieve of Eratosthenes (classical example found on [CHR08]).
`primes_swapped`(N)	4,800	3.4M	Variant of the previous handler, where non-tail recursion is used instead of tail recursion.
`ram_fib`(N)	300	20,000	Calculating N Fibonacci numbers using the RAM simulator (origin: [SSD05b]; see also [VWWSD08, Appendix A]), with the addition replaced by a multiplication to avoid arithmetic operations on large numbers (when using multiplication all Fibonacci numbers are equal to one).

Table 2.3: Limits for different CHR benchmark programs. The numbers in the second and third column indicate the approximate value of N for which the benchmark results in stack overflow for resp. JCHR and SWI; ∞ indicates the benchmark ran in constant stack space, and was thus only limited by heap space (none of the benchmarks reached this limit before timing out).

Chapter 3

Improved Compilation Scheme

This chapter introduces a new and improved compilation scheme for compiling CHR to Java. It is used by more recent versions of the K.U.Leuven JCHR system. In the traditional scheme for compiling CHR (cf. previous Chapter), the activation stack $\mathbb{A}$ corresponds to Java's implicit call stack. The direct adaption of the traditional compilation scheme to Java frequently leads to stack overflows, as seen in Section 2.3. The compilation scheme presented in this chapter overcomes this important issue by more explicitly maintaining the activation stack.

The structure of this chapter is analogous as before: building on the compilation scheme of Chapter 2, Section 3.1 presents a basic compilation scheme, which is subsequently optimized in Section 3.2, and evaluated in Section 3.3.

3.1 Basic Compilation Scheme

3.1.1 The `Handler` class

The `Handler` class (cf. Section 2.1.1) obtains the extra responsibility of managing a *continuation stack*. The abstract `Handler` is therefore extended with the code listed in Listing 3.1. Each handler implements the obvious `push` and `pop` methods to manipulate the continuation stack (lines 10–11). The `Continuation` class itself is a simple abstract class[1] (lines 1–3), with a single method `call()`.

[1] The reason it is not an interface is that in Java all methods of an interface are necessarily public. By making the `call()` method protected, some form of encapsulation is obtained: user code is never supposed to call the `call()` method. Also, Java folklore suggests calls to interfaces are slower than regular calls.

```
1  public abstract static class Continuation {
2      protected abstract void call();
3  }
4
5  protected void call(Continuation continuation) {
6      push(null);
7      do { continuation.call(); } while ((continuation = pop()) != null);
8  }
9
10 protected void push(Continuation continuation) { ... }
11 protected Continuation pop() { ... }
```

Listing 3.1: Code managing the continuation stack in the generated `Handler` classes.

Example 3.1 *Each* **`Constraint`** *class is a* **`Continuation`**. *Calling a* **`Constraint`** *continuation corresponds to calling the* `activate()` *method (cf. Chapter 2). In other words, a* **`Constraint`** *class overrides the abstract* `call` *method as follows:*

```
@Override
protected void call() { activate(); }
```

The loop on line 7 of Listing 3.1 now governs the control flow of a constraint handler. The enclosing `call(Continuation)` method (lines 5–8) is called with an initial continuation, typically a new constraint told from user code. First, `null` is pushed on the continuation stack, and then the initial continuation is called. During the execution of a continuation, other continuations may be pushed on the continuation stack. The loop on line 11 keeps popping and calling continuations, until the `null` value pushed on line 6 is reached. If this occurs, all work required for the initial continuation is done, and the method returns.

Example 3.2 *The* `tell`$C(\overline{args})$ *method for a constraint* $c(\overline{args})$ *is implemented as follows:*

```
public void tellC(args) {
    call(new CConstraint(args));
}
```

Calling the `activate()` *method is no longer done immediately, as was the case in Section 2.1.1, page 8. Instead, this is delegated to the* `call(Continuation)` *method (see Example 3.1).*

In general, it is possible that multiple `null` continuations appear on the continuation stack. This occurs for instance if a host language statement executed by the handler as part of a body adds new constraints to this handler using its `tell` methods. The continuation stack therefore actually represents multiple stacks, where each `null` value represents a stack's bottom.

3.1.2 The `Constraint` classes

In this section, all simple optimizations of Section 2.2 (except *Inlining*) are always applied immediately. Also the `activate()` method used in Example 3.2 is thus the version of Listing 2.5.

The basic compilation scheme for occurrence methods remains very similar to the scheme of Chapter 2 (Listings 2.2, 2.3, and 2.6). The main difference resides in the execution of (non-empty) rule bodies. To avoid problems with recursive stack overflows, instead of using the implicit JVM call stack, the continuation stack maintained by the enclosing `Handler` class is used. The scheme for **Simplify** transitions is discussed first (Section 3.1.2.1), followed by the slightly more complicated scheme for **Propagate** transitions (Section 3.1.2.2). These first two subsections furthermore assume bodies consist of CHR constraints only. Sections 3.1.2.3 and 3.1.2.4 then extend the scheme to deal with built-in constraints and host language statements respectively.

3.1.2.1 Simplify transitions (CHR constraints only)

Figure 3.1 illustrates the basic idea behind the new compilation scheme for **Simplify** transitions. Figure 3.1(a) recalls the relevant part of the compilation scheme of Chapter 2: all body conjuncts are called in conjunction, after which `false` is returned (see *Sequential Control Flow*, Section 2.2). Figure 3.1(b) illustrates what the stack-based JVM runtime actually does: before calling the first conjunct, a frame is pushed on a stack. After the execution of the first conjunct, this stack frame will be popped and called, executing the remainder of the body in the same manner. Calling a conjunct may of course push more frames on the call stack. In Java, this rapidly results in call stack overflows, certainly in the case of recursive CHR programs.

To solve this problem, the scheme in Figure 3.1(c) is used: instead of calling the first remaining conjunct, this conjunct is *also pushed* onto the continuation stack. The main control loop of Listing 3.1 will then pop this continuation and call it. After the execution of the first conjunct is completed, the continuation for the remainder of the body will be popped and called. By always

(a)	(b)	(c)

⋮
`b`$_1$`;`
`b`$_2$`; ...; b`$_{n_b}$`;`
 	
`return false;`
⋰
	⋮
	push`(`$\langle$`b`$_2$`,...,b`$_{n_b}$$\rangle$`);`
	call`(b`$_1$`);`
	
	`return false;`
	⋰
		⋮
		push`(`$\langle$`b`$_2$`,...,b`$_{n_b}$$\rangle$`);`
		push`(b`$_1$`);`
		
		`return false;`
		⋰

Figure 3.1: Pseudo-code compilation scheme for the execution of a body of a **Simplify** transition: (a) the original scheme (cf. Listing 2.3); (b) an illustrative scheme, faithfully simulating stack based execution; and (c) the actual (unoptimized) compilation scheme.

```
        ⋮
        push(new C_i_j_i_2(#vars(⟨b_2,...,b_n_b⟩)));
        push(new #constraint_type(b_1)(#args(b_1)));

        return false;
    }
  ⋰
```

Listing 3.2: Compilation scheme for a **Simplify** transition. This scheme replaces the corresponding code from Listing 2.3, as outlined in Figure 3.1 (if $n_b > 0$).

returning to the main control loop of Listing 3.1, recursion is essentially turned into iteration, thus solving the call stack overflow issues of the traditional compilation scheme.

Moving from the pseudo-code of Figure 3.1(c) to actual code is straightforward. For empty bodies the scheme of Listing 2.3 is simply kept, but for a non-empty body `b`$_1$`,...,b`$_{n_b}$, the scheme listed in Listing 3.2 is used. The following pseudo code operators are used:

- `#vars`: returns the variables used in an expression, in this case the variables used in the remainder of the body;
- `#constraint_type`: returns the name of the constraint class for the given constraint conjunct (recall that all body conjuncts, for now, are assumed to be CHR constraints); and
- `#args` returns the arguments of a constraint conjunct.

```
protected final class C_i_j_i_k extends Continuation {
    ...

    @Override
    protected void call() {
        push(new C_i_j_i_⟨k+1⟩(#vars(⟨b_k+1,...,b_n_b⟩)));
        push(new #constraint_type(b_k)(#args(b_k)));
    }
}
```

Listing 3.3: Compilation scheme for the continuations of Listings 3.2 and 3.5 in case $k \leq n_b$. The class also contains a constructor whose arguments are the variables used in the remainder of the body, and a series of member fields to store these variables after construction.

```
protected final class Ci_ji_⟨nb+1⟩ extends Continuation {
    @Override
    protected void call() { }
}
```

Listing 3.4: Compilation scheme for the last continuation of a **Simplify** transition.

```
        ⋮
        reactivated = false;

        store();

        push(new Ci_ji_2(#vars(⟨b2,...,bnb⟩), oldGeneration));
        push(new #constraint_type(b1)(#args(b1)));

        return false;
    }
⋰
```

Listing 3.5: Compilation scheme for a **Propagate** transition, replacing the corresponding code of Listing 2.6 (if $n_b > 0$).

The remainder of the body is executed in a similar fashion. For $k \leq n_b$, the $\mathtt{C}_i\mathtt{\_j}_i\mathtt{\_k}$ class (inner class of the $\mathtt{C}_i\mathtt{Constraint}$ class) implements the continuation for the remainder of the body starting with the k'th conjunct. The scheme for $k \leq n_b$ is shown in Listing 3.3; Listing 3.4 lists the trivial case for $k = n_b + 1$.

3.1.2.2 Propagate transitions (CHR constraints only)

The compilation of **Propagate** transitions is only slightly more complicated. After the execution of the body of the applied rule, more matching constraint combinations may have to be searched for that same rule (more precisely: for the *same occurrence*). If the body of the rule is empty, the old compilation scheme for **Propagate** transitions, Listing 2.6, remains unchanged for now. For non-empty bodies though, the scheme is adapted. The result is shown in Listing 3.5. The most important changes with Listing 2.6 are:

- Firstly, the body is of course executed using the explicit continuation stack. The basic compilation scheme for the continuations remains the same as for **Simplify** transitions, i.e. Listing 3.3. Only for the final continuation, Listing 3.4 is replaced with Listing 3.6. After the stepwise execution of the body, and if the active constraint is still alive (the constraint could have been removed indirectly by the execution of the body), Listing 3.6 resumes calling

```
protected final class Ci_ji_⟨nb+1⟩ extends Continuation {
  @Override
  protected void call() {
     if (isAlive() && !reactivated && ci-ji() && ci+1-ji+1() && ... && cni-jni())
        store();
  }
}
```

Listing 3.6: Compilation scheme for the last continuation of a **Propagate** transition.

the different occurrence methods (line 4). The sequence of occurrence methods called starts with the same occurrence method that was just applied. This guarantees that all matching constraint combinations will be found before advancing to the next occurrence (i.e., before a **Default** transition occurs). Recall that the propagation history will prevent the same constraint combination from being applied twice. Note that the *Drop after Reactivation* optimization of Section 2.2 is easily implemented as well (line 4).

- Secondly, even though it is not a dead continuation, the method returns `false` instead of `true` (Listing 3.5, line 9). The correspondence between the boolean that is returned, and the fact that the active constraint is alive or not, is thus lost. The interpretation of this boolean becomes as follows: `true` means the calling `activate()` or `reactivate()` method should continue with the next occurrence method (if there is one), and `false` means it should instead return to the outer control loop.

3.1.2.3 Built-in constraints

So far, the compilation scheme assumed bodies consisted of CHR constraints only. If the k'th body conjunct is a built-in constraint, the line:

```
push(new #constraint_type(b_k)(#args(b_k)));
```

in Listings 3.2, 3.3, and 3.5 is replaced with a call to a method of a built-in constraint solver. This causes the `reactivate()` method of all constraints that observe the variables involved to be called. Without proper care, this may cause stack overflows: these **Reactivate** transitions may cause more **Solve** transitions, which in turn may trigger more **Reactivate** transitions, and so on. The solution is to use the continuation stack for **Reactivate** transitions as well: all constraints that have to be reactivated are pushed on the continuation stack (recall that each constraint is a continuation). This way, the actual reactivation is postponed until the continuation is popped from the stack and called, thus avoiding stack overflows. Note though that this reverses the order in which constraints are reactivated. Indeed, the constraint that is pushed on the stack last is reactivated first. Reversing the reactivation order is allowed though, as the refined operational semantics does not determine the order in which constraints are put on the activation stack at a **Solve** transition.

The resulting code is shown in Listing 3.7:

Lines 1–8: By pushing the constraints to be reactivated on the continuation stack, reactivation is no longer performed by the `reactivate()` method, but through the `call()` method. For the *Drop after Reactivation* optimization of Section 2.2 to work, the `reactivated` flag has to be set to `true` after a constraint's reactivation. Note that the flag only has to set at a **Drop** transition, i.e. if all occurrences are traversed (lines 6 and 15). Because a reactivation may also end after a series of continuations, Listing 3.8 adds an extra line to Listing 3.6's compilation scheme for the last continuation of a **Propagate** transition.

The extra liveness test of line 3 is necessary because constraint could be terminated by reactivated constraints that were put on the stack later.

Lines 11–19: JCHR constraint solvers are supposed to be *incremental*. The expected semantics therefore entails that, immediately after a constraint is added, the previous solution is adapted to take into account the newly told constraint. A problem could occur when built-in constraints are told from host language code, be it as part of the initial query, or told during the execution of some host language statement in a rule's body. As incremental adaptation is expected, simply pushing the reactivations on a stack is then not acceptable. Indeed: the control would return to the host language code without reactivating any JCHR constraints, they would simply be pushed on some stack. Therefore, two modes are distinguished: if in *host language mode*, constraint reactivations are performed eagerly (lines 14–15), otherwise the constraints are pushed on the continuation stack (lines 16–17).

```
1  @Override
2  public void call() {
3     if (alive) {
4        if (occurrences()) {
5           store();
6           reactivated = true;
7        }
8     }
9  }
10
11 @Override
12 public void reactivate() {
13    if (inHostLanguageMode()) {
14       if (occurrences())
15          reactivated = true;
16    } else {
17       push(this);
18    }
19 }
```

Listing 3.7: Reactivation of constraints through the continuation stack. This code replaces the `reactivate()` method of Listing 2.5. The liveness test on line 3 is required in case of a reactivation (an earlier reactivation may have terminated the constraint).

```
protected final class C_i_j_i_<n_b+1> extends Continuation {
  @Override
  protected void call() {
     if (isAlive() && !reactivated && c_i-j_i() && c_{i+1}-j_{i+1}() && ... && c_{n_i}-j_{n_i}()) {
        store();
        reactivated = true;
     }
  }
}
```

Listing 3.8: Updated compilation scheme for the last continuation of a **Propagate** transition (replaces Listing 3.6).

```
protected void call(Continuation continuation) {
    setHostLanguageMode(false);
    push(null);
    do { continuation.call(); } while ((continuation = pop()) != null);
    setHostLanguageMode(true);
}
```

Listing 3.9: During CHR derivations the host language mode is, by default, switched off. This version of the `call(Continuation)` method replaces the one in Listing 3.1.

Initially, the constraint system is in host language mode. This is necessary for the initial query. During CHR derivations, the host language mode is, by default, switched off: see Listing 3.9. Only when control returns to host language code, the mode is set back to host language mode. This is explained further in Section 3.1.2.4.

3.1.2.4 Host language code

CHR rules can contain arbitrary host language statements. During the execution of such a statement, built-in or CHR constraints may be called from host language code. Recursion where CHR code is interleaved with host language code cannot be fully eliminated for incrementally adapting constraint solvers. As already seen in the previous subsection, this property implies constraints added from host language code should immediately adapt the solution, i.e., before returning from the method call. As this execution may recursively call the same host language statement, the only possibility to safeguard against stack overflows is to abandon incrementality.

By default, JCHR constraint solvers remain incremental, and stack overflow could occur as outlined above. However, JCHR offers the possibility to turn incrementality off. All built-in and CHR constraints told in host language mode are then queued, and pushed on the continuation stack in reversed order once control returns to the JCHR runtime.

If the k'th body conjunct is a host language statement, the line:

```
push(new #constraint_type(b_k)(#args(b_k)));
```

in Listings 3.2, 3.3, and 3.5 is thus replaced with:

```
setHostLanguageMode(true);
b_k;
setHostLanguageMode(false);
push(getQueue());
```

The queue is shared between all JCHR constraint handlers and built-in constraint solvers belonging to the same *constraint system.* This concept is introduced in the next section:

3.1.2.5 Constraint Systems

Because all built-in solvers and JCHR handlers need to be able to enqueue constraints added when in host language mode, some shared state is required. All cooperating solvers and handlers therefore are part of a *constraint system.* Built-in or CHR constraints must only range over either fixed values, or variables belonging to the same constraint system. This shared `ConstraintSystem` object manages the following state:

1. The constraint system's mode (i.e. host language mode or not, incremental mode or not). The different methods that inspect or modify this mode in a JCHR handler thus delegate to the handler's constraint system.
2. The constraint queue. The `tell`*`C`* methods responsible for adding constraints to a JCHR handler (cf. Example 3.2), as well as all procedures that add built-in constraints, have to be adjusted to queue constraints if the constraint system is in host language mode. The former adjustments are shown in the next subsection, the latter are outside the scope of this document.
3. The continuation stack: a set of related handlers also share a single continuation stack. Any stack-related method introduced in 3.1.1 thus delegates to the handler's `ConstraintSystem` as well. Section 3.1.2.6 will clarify why this is necessary.

3.1.2.6 JCHR handlers as built-in solver

JCHR allows to use a JCHR handler as a built-in solver for another JCHR handler These built-in constraints can then only be told (i.e. used in a body), and not asked (used in a guard). This may cause recursive relations between JCHR handlers. By definition these constraint handlers are part of the same constraint system (cf. previous subsection), and thus share the same continuation stack. This allows the last remaining risk of stack overflows to be avoided by replacing the `tellC` methods of Example 3.2 with:

```
public void tellC(args) {
    if (inHostLanguageMode()) {
        if (incrementalMode())
            call(new CConstraint(args));
        else
            enqueue(new CConstraint(args));
    } else {
        push(new CConstraint(args));
    }
}
```

A constraint system is not in host language mode if a built-in constraint is called. If therefore a built-in constraint is implemented by a JCHR handler, this constraint activation is pushed onto the shared continuation stack, thus avoiding call stack overflows due to recursive calls between JCHR handlers.

3.2 Optimizations

This section lists several optimizations applicable to the basic compilation scheme of the previous section. All optimizations of Section 2.2 are already applied. The optimizations introduced here are mostly aimed at reducing the constant time overheads incurred by creating explicit continuation objects and pushing them on a stack.

3.2.1 Trampoline Style Compilation

This important compilation scheme optimization is named after a popular technique to eliminate tail calls [TLA92, SO01]. A trampoline compilation scheme employs of an outer loop which repeatedly calls inner subroutines. Each time an inner subroutine wishes to tail call another subroutine, it does not call it directly, but simply returns a continuation to the outer loop, which then does the call itself. Tail recursive subroutines are therefore executed in constant stack space.

The compilation scheme presented here is similar, but more general: it also eliminates stack issues for non-tail calls. The optimized scheme nevertheless behaves exactly as a trampoline for bodies consisting of a single JCHR constraint—that is: tail recursive rules are guaranteed to execute in constant stack space. The analogy is strengthened further by other optimizations subsequent sections: the stack should only be used when really necessary, otherwise a trampoline style execution is used.

Trampoline Style Control Flow

In the stepwise execution of the body of the basic compilation scheme (cf. Listings 3.2–3.3, and 3.5) continuations are frequently popped immediately after they have been pushed. This is illustrated in Figure 3.2 to the left: right before the occurrence method or continuation call exits, the first conjunct of a remaining body is pushed onto the continuation stack. Immediately thereafter, that continuation is popped by the outer control loop. The *Trampoline Style Compilation* optimization aims at avoiding these superfluous (and relatively costly) `push` and `pop` operations.

The general idea, illustrated in Figure 3.2 to the right, is to return the next continuation directly to the outer control loop, instead of passing it via the continuation stack. For this purpose, the code in Listing 3.10 replaces the corresponding code in Listing 3.1. Calling a `Continuation` now always returns the next `Continuation` to call (line 2). The main control loop is adjusted

```
...
push(⟨continuation⟩);
push(⟨first conjunct⟩);
return ...;
...
pop();   ⇝ ⟨first conjunct⟩
...
```

$\Longrightarrow$

```
...
push(⟨continuation⟩);
return ⟨first conjunct⟩;
...
```

Figure 3.2: In the basic compilation scheme (Listings 3.2–3.3, 3.5), illustrated to the left, the first conjunct of the body is typically `push`ed, and then immediately `pop`ped again. The *Trampoline Style Compilation* optimization, illustrated to the right, solves this frequently recurring phenomenon by returning this continuation directly, instead of via the continuation stack.

```
public abstract static class Continuation {
    protected abstract Continuation call();
}

protected void call(Continuation continuation) {
    setHostLanguageMode(false);
    push(BOTTOM);
    while ((continuation = continuation.call()) != null);
    setHostLanguageMode(true);
}

private final static Continuation BOTTOM = new Continuation() {
    @Override
    public Continuation call() { return null; }
};
```

Listing 3.10: Trampoline style compilation scheme: main control loop (cf. Listings 3.1 and 3.9).

```
1  @Override
2  protected Continuation call() {
3     if (alive) {
4        Continuation continuation;
5        if ((continuation = c_1()) != null) return continuation;
6        ...
7        if ((continuation = c_n()) != null) return continuation;
8        store();
9        reactivated = true;
10    }
11    return pop();
12 }
```

Listing 3.11: The `call()` method of a constraint c with n occurrences using trampoline style control flow. This version replaces the one in Listing 3.7.

accordingly (line 8). Also, instead of pushing `null` to indicate the bottom of a (virtual) stack (cf. Section 3.1.1, page 23), a special `BOTTOM` continuation is used (lines 7, 12–15). The reason for this will be clarified shortly. As this `BOTTOM` continuation is the only continuation that is allowed to return `null` when its `call()` method is invoked, the main loop can still easily detected when the bottom of a stack is reached.

The `call()` method of a constraint becomes implemented as in Listing 3.11. Note that the `activate()` now is no longer used, and can be removed. The compilation scheme combines the ideas of trampoline style control flow with those of the sequential control flow introduced in Section 2.2. As before, all occurrence methods are called in sequence. Only, instead of returning a boolean, occurrence methods now return a `Continuation`. Instead of returning `false`, the next continuation to execute will be returned. If this continuation is a conjunct of the body, which is the most common case, this saves pushing the continuation on the stack first (cf. Figure 3.1). This was the central idea of this optimization. If the there is no remaining body conjunct, the same two cases as before are considered:

Dead Continuations If a **Drop** transition occurs (line 9), or similarly at the end of a **Simplify** transition (Listing 3.2(c)), the next continuation is popped from the continuation stack and returned.

Live Continuations To indicate the next occurrence method should be tried, `null` is returned instead of `true`. This is why the `BOTTOM` continuation was introduced before: it allows to distinguish between a live continuation (`null`), and the case where the bottom of the stack is reached (`BOTTOM`).

Note that line 7 of Listing 3.11 implements the *Late Storage* optimization, and line 8 is required for the *Drop after Reactivation* optimization (as in Listing 3.7).

The compilation schemes for the occurrence methods and their associated continuation classes is listed in Listing 3.12. As in Section 3.1, this scheme deals with bodies consisting of CHR constraints only. The next subsection addresses built-in constraints and host language statements.

Built-in Constraints and Host Language Statements

Built-in Constraints If the k'th body conjunct is a built-in constraint, the compilation scheme proposed in Section 3.1.2.3 is:

```
...
push(new C_i_j_i_⟨k+1⟩(#vars(⟨b_{k+1},...,b_{n_b}⟩), ...));
b_k();
return false;
```

```
        ⋮
      push(new Ci_ji_2(#vars(⟨b2,...,bnb⟩)));
      return new #constraint_type(b1)(#args(b1));
   }
⋰
```

```
        ⋮
      return pop();
   }
⋰
```

```
        ⋮
      continue label_cr-jr;
   }
⋰
```

(a) The above, left scheme replaces Listings 3.2 and 3.5 for a non-empty body. For an empty rule body, Listing 3.2 for **Simplify** transitions is replaced with the top right scheme, and Listing 3.5 for **Propagate** transitions with the bottom right one.

```
protected final class Ci_ji_k extends Continuation {
   ...

   @Override
   protected Continuation call() {
      push(new Ci_ji_⟨k+1⟩(#vars(⟨bk+1,...,bnb⟩)));
      return new #constraint_type(bk)(#args(bk));
   }
}
```

(b) Scheme for continuations with $k \leq n_b$ (replaces Listing 3.3).

```
protected final class Ci_ji_⟨nb+1⟩ extends Continuation {
    @Override
    protected Continuation call() { return pop(); }
}
```

(c) Scheme for the last continuation of a **Simplify** transition (replaces Listing 3.4).

```
protected final class Ci_ji_⟨nb+1⟩ extends Continuation {
   ...

   @Override
   protected Continuation call() {
      if (isAlive() && !reactivated) {
         Continuation continuation;
         if ((continuation = ci-ji()) != null) return continuation;
         if ((continuation = ci+1-ji+1()) != null) return continuation;
         ...
         if ((continuation = cni-jni()) != null) return continuation;
         store();
         reactivated = true;
      }
      return pop();
   }
}
```

(d) Scheme for the last continuation of a **Propagate** transition (replaces Listing 3.6).

Listing 3.12: Trampoline style compilation scheme: optimizations of Listings 3.2–3.6.

Firstly, this scheme has to be adapted to the trampoline style compilation scheme introduced by the previous optimization. Secondly, a further optimization is possible. The built-in constraint call '$\texttt{b}_k\texttt{();}$' does not always push reactivations onto the stack. In that case creating, pushing and popping the $\texttt{C}_i\texttt{\_j}_i\texttt{\_}\langle\texttt{k+1}\rangle$ continuation, as well as the possible overhead of resuming the search for partner constraints, may be avoided.

The resulting compilation scheme for a series of built-in constraint calls is listed in Listing 3.13. The `swap(Continuation, int)` operation replaces the continuation on a given stack depth, and returns the continuation formerly on that depth. This way, the first reactivation pushed is called first (and the others are called in reverse order, cf. Section 3.1.2.3). Note that this scheme also deals with built-in constraints implemented using JCHR, as described in Section 3.1.2.6.

Host Language Statements The compilation scheme for host language statements is similar. The original scheme, introduced in Section 3.1.2.4, is as follows:

```
...
push(new C_i_j_i_⟨k+1⟩(#vars(⟨b_{k+1},...,b_{n_b}⟩), ...));
setHostLanguageMode(true);
b_k();
setHostLanguageMode(false);
push(getQueue());
return false;
```

However, even if incrementality is turned off (cf. Section 3.1.2.4), the queue will mostly be empty. So there is normally no reason to `push` and `return` after each host language statement. The optimized scheme for a series of host-language statements $\texttt{b}_k\texttt{, b}_{k+1}\texttt{, ..., b}_{\kappa-1}$ is listed in Listing 3.14. The convenience method `dequeue(Continuation)` performs the following steps:

1. it sets the host language mode to `false`;
2. it pushes the provided continuation;
3. it pushes all but the first queued continuation onto the stack, in the reverse order in which they were queued; and
4. it returns the continuation that was queued first.

Interleaving Of course, built-in constraints and host language statements can be interleaved arbitrarily. Combining the compilation schemes proposed in Listings 3.13 and 3.14 for the cases where the first part of the (remaining) body is an interleaving of built-in constraints and host language statements is straightforward. The main idea is that the creation of continuations, and the use of the continuation stack, is avoided as much as possible.

3.2.2 Generic Optimizations

This section presents a series of generic optimizations. Optimizations that are specific to **Simplify** and **Propagate** transitions are considered in Sections 3.2.3 and 3.2.4 respectively.

Drop *before* Reactivation

Section 3.1.2.3 explained that if a **Solve** transition triggers **Reactivate** transitions, corresponding reactivate continuations are pushed onto the continuation stack. During the execution of one of these continuations, more reactivation continuations may be pushed onto the stack. This way, it is possible that there are multiple reactivation continuations on the stack for a single constraint. Following the same reasoning (and proof: cf. [Sch05a]) as with the *Drop after Reactivation* optimization of Section 2.2, only the top most of these reactivations has to be executed. Therefore, the scheme of Listing 3.11 is replaced with that of Listing 3.15 (the only difference is the additional

```
1   ...
2   final int SS = getStackSize();
3   b_k();
4   if (getStackSize() != stackSize)
5       return swap(new C_i_j_i_⟨k+1⟩(#vars(⟨b_{k+1},...,b_{n_b}⟩)), SS);
6   ...
7   b_{κ−1}();
8   if (getStackSize() != stackSize)
9       return swap(new C_i_j_i_κ(#vars(⟨b_κ,...,b_{n_b}⟩)), SS);
10  push(new C_i_j_i_⟨κ+1⟩(#vars(⟨b_{κ+1},...,b_{n_b}⟩)));
11  return new #constraint_type(b_κ)(#args(b_κ));
12 ⋰
```

Listing 3.13: Optimized, trampoline style compilation scheme for bodies containing built-in constraints. This scheme replaces the corresponding code in Listing 3.12(a)–(b) if the first conjuncts of the (remaining) body are built-in constraints $\texttt{b}_k$, $\texttt{b}_{k+1}$, ..., $\texttt{b}_{\kappa-1}$, with $\kappa > k$ the index of the first JCHR constraint in the body after $\texttt{b}_k$. If there is no such JCHR constraint left in the body, one option would be to replace lines 10–11 with "`return new` $\texttt{C}_i\texttt{\_j}_i\texttt{\_}\langle n_b + 1\rangle$`();`". As this continuation is actually unnecessary (recall the goal of this optimization is to reduce the number of continuations and stack operations), the body of its `call()` method is simply inlined instead: see Listing 3.12(c) or (d), depending on whether it concerns a **Simplify** or **Propagate** transition.

```
    ...
    setHostLanguageMode(true);
    b_k();
    if (hasQueued())
        return dequeue(new C_i_j_i_⟨k+1⟩(#vars(⟨b_{k+1},...,b_{n_b}⟩));
    ...
    b_{κ−1}();
    if (hasQueued())
        return dequeue(new C_i_j_i_κ(#vars(⟨b_κ,...,b_{n_b}⟩));
    setHostLanguageMode(false);
    push(new C_i_j_i_⟨κ+1⟩(#vars(⟨b_{κ+1},...,b_{n_b}⟩));
    return new #constraint_type(b_κ)(#args(b_κ));
⋰
```

Listing 3.14: Optimized, trampoline style compilation scheme for bodies containing host language code. This scheme replaces the corresponding code in Listing 3.12(a)–(b) if the first conjuncts of the (remaining) body are a series of host language statements $\texttt{b}_k$, $\texttt{b}_{k+1}$, ..., $\texttt{b}_{\kappa-1}$, with $\kappa > k$ the index of the first JCHR constraint in the body after $\texttt{b}_k$. Analogously to the scheme in Listing 3.13, if there is no such JCHR constraint left in the body, lines 10–11 are replaced with the body of the `call()` method of either Listing 3.12(c) (for a **Simplify** transition) or Listing 3.12(d) (for a **Propagate** transition).

```
1  @Override
2  protected Continuation call() {
3     if (alive && !reactivated) {
4        Continuation continuation;
5        if ((continuation = c_1()) != null) return continuation;
6        ...
7        if ((continuation = c_n()) != null) return continuation;
8        store();
9        reactivated = true;
10    }
11    return pop();
12 }
```

Listing 3.15: The `call()` method of a C`Constraint` class after the *Drop before Reactivation* optimization. This version replaces the one in Listing 3.11 (the only difference is the additional test on line 3).

test on line 3). Analogously to the earlier *Drop after Reactivation* optimization, this optimization is again only applied if at least one of the occurrences in the rule's head may be reactivated.

Eager Pushing

The stepwise execution of the body in the basic compilation scheme is illustrated in Figure 3.3 to the left. At lines $2a$, $7a$, and $12a$, special continuation objects are created to represent the remainder of the body, and pushed on the stack. However, as illustrated to the right, it is often possible to push the different conjuncts of the body eagerly at the moment the body is first applied (line $2b$), thus saving the creation of the latter continuation objects (the constraint objects pushed on line $2b$ have to be created anyway, namely on lines $8a$ and $13a$). Less `Continuation` classes have to be generated as well. Some more method calls are avoided by generating specialized `push` operations for pushing multiple continuations at once (line $2b$).

This optimization is not always applicable. If for instance some conjunct requires the execution of a previous conjunct to ground a variable (this depends on the type or mode declaration of the constraints), it is not be possible to create and push the former before the latter is executed.

Traditional Compilation

If it can be shown that calling a certain constraint can never overflow the call stack, the scheme of Chapter 2 can safely used. This will most likely improve performance, since maintaining the call stack explicitly will always involve some constant time overhead (see also Chapter 3.3).

The problem at hand is thus to show that activating a constraint only requires a small, fixed number of stack frames. This is clearly the case if the activation of this constraint never, directly or indirectly, encounters recursive constraint calls. One technique that can be used is the computation of the transitive closure of a constraint call graph. More advanced analysis techniques, such as abstract interpretation (see [SSD05a]), can also be used. Details are outside the scope of this paper.

3.2.3 Optimizing Simplify Transitions

For a **Simplify** transition the active constraint is terminated. Consequently, after the body is completely executed, no more work needs to be done[2]. Nevertheless, in the basic compilation scheme,

[2] If functional dependency or set semantics information is available (see e.g. [HGSD05, DS05]), kept occurrences for which all partner constraints are unique may be treated analogously to removed occurrences.

```
1a   ...
2a   push(new Continuation⟨b2, b3⟩);
3a   return b1;
4a   ...
5a   return pop(); ⇝ ⟨b2, b3⟩
6a   ...
7a   push(new Continuation⟨b3⟩);
8a   return b2;
9a   ...
10a  return pop(); ⇝ ⟨b3⟩
11a  ...
12a  push(new Continuation⟨⟩);
13a  return b3;
14a  ...
15a  return pop(); ⇝ ⟨⟩
16a  ...
```

$\Longrightarrow$

```
1b   ...
2b   push(b2, b3, new Continuation⟨⟩);
3b   return b1;
4b   ...
5b   return pop(); ⇝ b2
6b   ...
7b   return pop(); ⇝ b3
8b   ...
9b   return pop(); ⇝ ⟨⟩
10b  ...
```

Figure 3.3: The *Eager Pushing* optimization: by eagerly pushing the conjuncts of the body (line 2*b*), less continuation objects have to created (lines 2*a* and 7*a*). The illustrated example is for '$\mathtt{b}_1, \mathtt{b}_2, \mathtt{b}_3$', a body with three JCHR constraint conjuncts.

trivial continuations are pushed for this (see Listings 3.4 and 3.12(c), and also the illustration in Figure 3.3). In the optimized scheme, these 'empty' continuations are not pushed.

A similar optimization is also possible in the compilation schemes for built-in constraints and host language statements presented earlier in this section. For **Simplify** transitions, if a *built-in constraint* is the last conjunct of the body, pushing a continuation is again pointless, as no more work needs to be done for the active constraint. Consequently, the stack size does not have to be compared. The '`return pop();`' operation that follows suffices, even if reactivations were pushed on stack by the last conjunct. Similarly, if the last conjunct of a `Simplify` transition is a *host language statement*, the statement following this host language statement becomes:

```
return hasQueued()? dequeue() : pop();
```

3.2.4 Optimizing Propagate Transitions

Resuming the Search for Partner Constraints

After the (non-empty) body of a **Propagate** transition is completely executed, Listing 3.12(d) starts by calling the same occurrence method again. Because either one or more partner constraints is removed (simpagation rules), or it is prevented by the propagation history (propagation rules), a rule will never be applied with the same constraint combination more than once. Nevertheless, each time all iterators of the nested loops are restarted, generally recomputing the same partial joins again as before[3], up to the point where the last **Propagate** transition was applied. In other words: a significant amount of redundant work is done over and over again during a consecutive series of **Propagate** transitions.

The solution is to include all iterators in the continuation, and use them to resume the search for partner constraints directly at the point where the search was interrupted. We first consider **Propagate** transitions with simpagation rules. Afterwards, the scheme is easily adjusted for propagation rules.

[3] Often even additional, redundant (partial) joins are computed, as executing the body may have added matching constraints. As these have already been active, they will most likely not match again.

```
1          ⋮
2          push(
3             new #constraint_type(b_{k+1})(#args(b_{k+1})),
4             ...,
5             new #constraint_type(b_{k+l})(#args(b_{k+l})),
6             new C_i_j_i_⟨k+1⟩(
7                #vars(⟨b_{k+l+1},...,b_{n_b}⟩),
8                c_1_j_1, c_1_j_1_iter, ..., c_{r-1}_j_{r-1}, c_{r-1}_j_{r-1}_iter, c_r_j_r_iter
9             )
10         );
11         return new #constraint_type(b_k)(#args(b_k));
12      }
13   ⋰
```

Listing 3.16: Compilation scheme for a **Propagate** transition with a multi-headed simpagation rule (for which $n_b > 0$), after application of the *Eager Pushing* and *Live Continuations* optimizations. This scheme replaces the corresponding code in Listing 3.12(a) ($\rightsquigarrow k = 0$), and Listing 3.12(b) ($\rightsquigarrow 0 < k \leq n_b$).

Simpagation Rules Listing 3.16 lists the compilation scheme for pushing the continuations for **Propagate** transitions with a simpagation rule. We assumed the first remaining body conjunct, b_k, is a JCHR constraint. If this is not the case, the scheme is adjusted as shown earlier in Listings 3.13 and 3.14. We furthermore assumed the l conjuncts after b_k are JCHR constraints that can be pushed eagerly (see the *Eager Pushing* optimization), though l may be zero. Next, a continuation is pushed. The arguments passed to it on lines 7–8 are:

- The variables used in the remainder of the body.
- The different iterator objects needed to resume the search for partner constraints after the body if fully executed. Only the iterators of the kept occurrences are passed, as well as the iterator of the first removed occurrence. This is because the latter is the iterator that has to advanced next (cf. the *Explicit Backjumping* optimization in Section 2.2). Any more deeply nested iterators will then be reinitialized, so there is no need to pass them to the continuation.
- The current partner constraints for all kept occurrences, as these may be required for propagation histories or testing for mutually different partner constraints (cf. for instance line 34 in Listing 2.2).

The `C`$_i$`_j`$_i$`_`⟨`n`$_b$`+1`⟩ continuation class, i.e. the class for the last continuation of a **Propagate** transition, is adjusted as shown in Listing 3.17. Instead of calling the same occurrence method again, as in Listing 3.12(d), a special occurrence method is invoked with the necessary arguments (lines 9–11). The compilation scheme of this method is sketched in Listing 3.18, and is similar to that of an ordinary occurrence method (cf. Listing 2.2). A boolean `first` is used though to ensure the search is resumed correctly. In the most common case—ignore lines 7, 13, and 17 for now—the search is resumed by advancing the iterator of the first removed occurrence. On line 6, the boolean `first` is initialized to `true`. This boolean is used to ensure that *the first time* none of the outer iterators are advanced (lines 9, 15, and 19), and that none of the more outer iterators are reinitialized (lines 11 and 21). The first iterator that is advanced is the one for the first removed occurrence, on line 25. Afterwards, the nested loops continue as before, since the `first` boolean is always `true` after line 22 is reached the first time. Note that if there is only one partner constraint, there is no need to introduce the boolean `first`, as the search will always have to resume by advancing the first iterator.

```
1  protected final class C_{i}_j_{i}_⟨n_{b}+1⟩ extends Continuation {
2     ...
3  
4     @Override
5     protected Continuation call() {
6        ...
7        if (isAlive() && !reactivated) {
8           Continuation continuation;
9           if ((continuation = c_{i}_j_{i}_⟨n_{b}+1⟩(
10                c_{1}_j_{1},c_{1}_j_{1}_iter,...,c_{r-1}_j_{r-1},c_{r-1}_j_{r-1}_iter,c_{r}_j_{r}_iter
11          )) != null) return continuation;
12          if ((continuation = c_{i+1}_j_{i+1}()) != null) return continuation;
13          ...
14          if ((continuation = c_{n_i}_j_{n_i}()) != null) return continuation;
15          store();
16          reactivated = true;
17       }
18       return pop();
19    }
20 }
```

Listing 3.17: Compilation scheme for the last continuation of a **Propagate** transition with the *Live Continuations* optimization applied. This scheme replaces Listing 3.12(d). Both the constructor and the member fields are extended to incorporate the iterators and current partner constraints required on line 10. If the last part of the body consists of built-in constraints or host language statements, line 6 contains the execution of these body conjuncts (following the compilation scheme of Listings 3.13 and 3.14).

```
1  private final Continuation c_i_j_i_⟨n_b+1⟩(
2      #constraint_type(c_1) c_1_j_1, Iterator<#constraint_type(c_1)> c_1_j_1_iter, ...,
3      #constraint_type(c_{r-1}) c_{r-1}_j_{r-1}, Iterator<#constraint_type(c_{r-1})> c_{r-1}_j_{r-1}_iter,
4      Iterator<#constraint_type(c_r)> c_r_j_r_iter
5  ) {
6     boolean first = true;
7     if (!c_1_j_1.isAlive()) first = false;
8     while (first || c_1_j_1_iter.hasNext()) {
9        if (!first) c_1_j_1 = c_1_j_1_iter.next();
10        ⋮
11       if (!first) c_2_j_2_iter = lookupC_2();
12        ⋮
13       if (!c_2_j_2.isAlive()) first = false;
14       while (first || c_2_j_2_iter.hasNext()) {
15          if (!first) c_2_j_2 = c_1_j_1_iter.next();
16             ⋱
17                if (!c_{r-1}_j_{r-1}.isAlive()) first = false;
18                while (first || c_{r-1}_j_{r-1}_iter.hasNext()) {
19                   if (!first) c_{r-1}_j_{r-1} = c_{r-1}_j_{r-1}_iter.next();
20                    ⋮
21                   if (!first) c_r_j_r_iter = lookupC_r();
22                   else first = false;
23                    ⋮
24                   while (c_r_j_r_iter.hasNext()) {
25                      #constraint_type(c_r) c_r_j_r = c_r_j_r_iter.next();
26                       ⋮
27                      Iterator<#constraint_type(c_{r+1})> c_{r+1}_j_{r+1}_iter = lookupC_{r+1}();
28                         ⋱
```

Listing 3.18: Resuming the search for partner constraints after a **Propagate** transition for a simplification rule with more than two heads. Only the code relevant to the search for partner constraints is shown. The remaining code is compiled analogously to ordinary occurrence methods (including all applicable optimizations). Note that the scheme contains the equivalent of the *Backjumping* optimization of Section 2.2 (lines 7, 13, and 17).

The equivalent of the *Backjumping* optimization of Section 2.2 is incorporated as well. As already explained, by default, the iteration of the first removed partner is resumed first. The *Backjumping* optimization though requires the search to be resumed in a more outer loop if the corresponding partner constraint was terminated by executing the body in the previous **Propagate** transition. This is implemented by lines 7, 13, and 17: by setting `first` to `false` early, the more outer loops are resumed if necessary.

The scheme of Listing 3.18 only shows the parts relevant to resuming the search for partner constraints. The remaining code is analogous to that for regular **Propagate** transitions. One observation though: as this continuation continues after the application of an earlier **Propagate** transition, late storage no longer has to be applied, as the active constraint will have been stored prior to the earlier rule application. In other words, line 4 of Listing 3.16 can be omitted in the remainder of Listing 3.18.

Propagation Rules Listings 3.16–3.18 only showed the case for simplification rules. The scheme is easily adjusted to propagation rules though:

- The case of single-headed propagation rules is treated in the next optimization.
- In case of a propagation rule with more than two heads, `first` is set to `false` right before the last kept partner (instead of before the first removed partner). In Listings 3.16–3.17, all iterators are passed, and all current partner constraints except the last partner (as this iterator will be advanced immediately).

Next Occurrence

By the previous optimization, a continuation is pushed to resume the search for matching constraint combinations after a **Propagate** transition. For single-headed propagation rules, no partner constraints have to be searched[4], and Listing 3.18 reduces to a trivial method with body '`return null;`'. The `push` statement on lines 2–10 of Listing 3.16 can therefore be replaced with:

```
push(
   new #constraint_type(b_{k+1})(#args(b_{k+1})),
   ...,
   new #constraint_type(b_{k+l})(#args(b_{k+l})),
   new C_i_⟨j_i+1⟩(#vars(⟨b_{k+l+1},...,b_{n_b}⟩))
);
```

Here `C_⟨j+1⟩` is a new `Continuation` class, very similar to the `C_j_⟨n_b+1⟩` class of Listing 3.17. The only difference is that, after executing the remainder of the body (if present), `C_⟨j+1⟩` starts with calling the $\langle j_i + 1\rangle$'th occurrence method. If there is no $\langle j_i + 1\rangle$'th occurrence, the above `push` statement can further be simplified to:

```
push(
   new #constraint_type(b_{k+1})(#args(b_{k+1})),
   ...,
   new #constraint_type(b_{k+l})(#args(b_{k+l}))
);
store();
reactivated = true;
```

Note that the last two lines are executed now a bit sooner then before, but this is no perfectly allowed. Analogous optimizations are also applied for the compilation schemes of built-in constraints and host language statements in the body of rules (cf. Listings 3.13 and 3.14).

[4] If information on set semantics is available (see [HGSD05, DS05]), the *Next Occurrence* optimization can also be applied if all partners of the kept occurrence are known to be unique.

Early Testing

For kept occurrences with only a single partner constraint[5], resuming the search for partner constraints may sometimes be avoided. If *partner*_iter, the constraint iterator for the single partner constraint, is known never to return constraints added after its creation (recall from Section 2.1.1, page 8, that this is allowed), the push statement on lines 2–10 of Listing 3.16 can be replaced with:

```
if (partner_iter.hasNext())
   push(
      new #constraint_type(b_{k+1})(#args(b_{k+1})),
      ...,
      new #constraint_type(b_{k+l})(#args(b_{k+l})),
      new C_i_j_i_⟨k+1⟩(
         #vars(⟨b_{k+l+1},...,b_{n_b}⟩),
         c_1_j_1, c_1_j_1_iter, ..., c_{r-1}_j_{r-1}, c_{r-1}_j_{r-1}_iter, c_r_j_r_iter
      )
   );
else
   push(
      new #constraint_type(b_{k+1})(#args(b_{k+1})),
      ...,
      new #constraint_type(b_{k+l})(#args(b_{k+l})),
      new C_i_⟨j_i+1⟩(#vars(⟨b_{k+l+1},...,b_{n_b}⟩))
   );
```

The push statement in the **else**-branch is analogous to the one proposed in the *Next Occurrence* optimization. So, if there is no $\langle j_i + 1\rangle$'th occurrence, the else-branch can again be simplified as indicated there.

Suppose *partner*_iter.hasNext() returns true, and a continuation is pushed. If analysis shows that executing the body cannot remove constraints of the partner constraint's type, resuming the search for partner constraints afterwards does not have to start with a call to hasNext(). In other words: the while loop of Listing 3.18 can be replaced with a do-while loop.

Lazy Popping

This last subsection introduces yet another optimization for non-removed active constraints that only look up a single partner constraint. The optimization is illustrated in Figure 3.4. If the body consists of a single JCHR constraint (b_x in Figure 3.4), an identical continuation is pushed over and over again, once for each **Propagate** transition (lines 2,9,16,... in the figure).

In the optimized compilation scheme, this continuation is only pushed once (line 2). The remaining push operations are replaced with undoPop operations (lines 9,16,...). The undoPop operation restores the stack to its state prior to the previous pop, i.e. re-adds the previous continuation to the stack. As it is sufficient that one pop operation can be undone, the undoPop operation can be implemented very efficiently. This optimization saves the creation of many identical continuation objects.

The optimization remains applicable if the body also contains built-in or host language statements, as long as these do not cause other continuations to be pushed.

3.3 Evaluation

To verify our implementation's competitiveness, we benchmarked the performance of some typical CHR programs using sever state-of-the-art CHR implementations (the same as used in Section 2.3).

[5] The optimization can be generalized if set semantics information is derived (see [HGSD05, DS05]): *Early Testing* can then also be applied if there is only one non-unique partner constraint.

```
...
push(new Continuation⟨vars , partner_iter⟩);
return b_x;
...
return pop(); ⇝ ⟨vars , partner_iter⟩
...
// resume search for partner constraint + fire again with different partner
...
~~push(new Continuation⟨vars , partner_iter⟩);~~ ⟹ undoPop();
return b_x;
...
return pop(); ⇝ ⟨vars , partner_iter⟩
...
// resume search for partner constraints + fire again with different partner
...
~~push(new Continuation⟨vars , partner_iter⟩);~~ ⟹ undoPop();
return b_x;
...
return pop(); ⇝ ⟨vars , partner_iter⟩
...
```

Figure 3.4: The *Lazy Popping* optimization: the last continuation popped is remembered, and can be restored with the `undoPop()` operation.

	tak(500, 450, 405)		dijkstra(16, 384)		leq(100)		ram_fib(N) $N = 25k$		ram_fib(N) $N = 200k$	
YAP	2,310	(100%)	44,000	(100%)	4,110	(100%)	1,760	(100%)	15,700	(100%)
SWI	3,930	(170%)	6,620	(15%)	17,800	(433%)	1,000	(57%)	*stack overflow*	
CCHR	48	(2.1%)	1,170	(2.7%)	189	(4.5%)	416	(24%)	3,540	(23%)
JCHR	183	(7.9%)	704	(1.6%)	68	(1.7%)	157	(8.9%)	1,714	(11%)
C	10	(0.4%)	-		2	(.05%)	1.3	(.07%)	12.7	(.08%)
Java	11	(0.5%)	-		2	(.05%)	2	(.11%)	16	(.10%)

Table 3.1: Benchmark comparing performance in some typical CHR programs in several systems. The average CPU runtime in milliseconds is given and, between parentheses, the relative performance with YAP Prolog as the reference system.

Execution times for native implementations in C and Java were added for reference. The results[6] are found in Table 3.1. The `dijkstra` and `ram_fib` benchmarks were described earlier in Table 2.3. The `tak` benchmark evaluates the well-known Takeuchi function (with tabling), and the `leq`(N) benchmark is a classic CHR benchmark that solves a circular list of N less-or-equal constraints.

As seen in Section 2.3.1, the SWI runtime does not perform the necessary tail call optimizations for the `ram_fib` benchmark. For 200k Fibonacci numbers, this benchmark therefore results in a stack overflow. Both JCHR and CCHR use the optimized compilation scheme presented in this chapter. Recall from Section 2.3.1 that, using the traditional compilation scheme, JCHR already incurred call stack overflows for `dijkstra`(2,100) and `ram_fib`(300). Using the optimized compilation scheme, recursive JCHR handlers become only limited by available heap space. From Table 3.1 it is clear that this more than sufficient.

The imperative CHR systems are significantly faster than both Prolog systems, up to one or

[6] Benchmark results of Table 3.1 are taken from [VWWSD08]. The benchmarks were performed on a Intel® Core™ 2 Duo 6400 system with 2 GiB of RAM. SWI-Prolog 5.6.50 and YAP 5.1.2 were used. All C programs were compiled with GCC 4.1.3. K.U.Leuven JCHR 1.6.0 was used; Java code was compiled with Sun's JDK 1.6.0 and executed with HotSpot JRE 1.6.0. The source code of the benchmarks used is available at http://www.cs.kuleuven.be/~petervw/bench/lnai2008/

	Traditional scheme	Improved scheme *unoptimized*		Improved scheme *optimized*	
`beer`(2,000)	7,553	7,341	(-3%)	7,058	(-7%/-4%)
`bool`(1,000,000)	4,106	5,216	(+27%)	4,914	(+20%/-6%)
`dijkstra`(5,000)	*stack overflow*	1,230		1,065	(-13%)
`fib`(33)	9,366	12,816	(+37%)	10,934	(+17%/-15%)
`gcd`(64,000,000)	*stack overflow*	5,529		5,519	(-0%)
`leq`(300)	3,821	5,537	(+45%)	4,309	(+13%/-22%)
`mergesort`(100,000)	10,581	12,262	(+16%)	11,310	(+7%/-8%)
`primes`(10,000)	*stack overflow*	4,991		4,396	(-13%)
`union`(210,000)	2,637	2,685	(+2%)	2,637	(+0%/-2%)
`ram_fib`(150,000)	*stack overflow*	3,144		2,991	(-5%)

Table 3.2: Empirical comparison between the different compilation schemes. The first column gives timings (in average milliseconds) when using the traditional compilation scheme of Chapter 2. For the remaining columns, this chapter's compilation scheme was used. The second column gives the results when the optimizations of Section 3.2 were not applied[7], the fourth when they were. The percentages between parentheses give the relative difference with the traditional compilation scheme (if applicable), and in the case of the last column, also the relative difference between the unoptimized and the optimized version of the improved compilation scheme.

two orders of magnitude, depending on the benchmark. This is partly due to the fact that the generated Java and C code is (just-in-time) compiled, whereas the Prolog code is interpreted. The native C and Java implementations remain two orders of magnitude faster than their CHR counterparts. The main reason is that these programs use specialized, low-level data structures, or exploit domain knowledge difficult to derive from the CHR program. The Dijkstra algorithm was not implemented natively.

Finally, we ran a number of benchmarks to compare the optimized compilation scheme against the traditional one, and to evaluate the optimizations listed in Section 3.2. The results[8] are listed in Table 3.2. It is clearly seen that the improved compilation scheme no longer results in stack overflows. Explicitly maintaining the call stack, however, is probably inherently more expensive in Java than relying on the JVM's implicit call stack. It is therefore to be expected that the improved compilation scheme is less efficient than the traditional one, provided the latter does not result in a stack overflow. The results in Table 3.2 confirm this. The optimizations of Section 3.2, however, are capable of considerably reducing the stack's overhead. The optimized improved compilation scheme is never more than 20% slower than the traditional scheme, and in many cases it becomes equally fast, or just a few percent slower. For one benchmark, the improved scheme is even faster than the traditional one.

The benchmark results thus show that, for compiling CHR to imperative languages, and in particular to Java, our new, improved compilation scheme is superior to the traditional one. All stack overflow issues are resolved, and our optimizations reduce the overhead to an acceptable level.

[7] Trampoline style compilation (Section 3.2.1) is always used, and iterators are always included in the continuations for resuming the search for partner constraints (Section 3.2.4). These only listed as optimizations in this report for presentation purposes, and cannot be switched off in the current implementation.

[8] The benchmarks of Table 2.3 were performed on a Intel®Pentium®4 CPU 2GHz with 1GiB of RAM. K.U.Leuven JCHR 1.6.1 and SWI-Prolog 5.6.55 were used. The Java code was compiled with Sun's JDK 1.6.0 and executed with HotSpot JRE 1.6.0.

Chapter 4

Conclusions

In this report, we reconstructed the compilation scheme used by the K.U.Leuven JCHR system [VSD05, VW08a] to compile CHR handlers to efficient Java code. Starting from basic compilation schemes, we gradually introduced several important optimizations performed by the compiler.

Two compilation schemes are presented. The traditional scheme, used by earlier versions of JCHR [VW05], is a relatively straightforward adaptation of the compilation scheme used by most Prolog embeddings of CHR. Practice, however, revealed that this scheme is less suited for imperative host languages. The reason is that imperative languages, such as Java, commonly do not perform the necessary recursion optimizations required to avoid call stack overflows. Executing CHR programs compiled using the traditional CHR compilation scheme therefore frequently results in fatal call stack overflows.

We therefore designed a new and improved compilation scheme, that explicitly manages a continuation stack. Using the new compilation scheme, CHR handlers no longer cause stack overflows. Next, we introduced several optimizations to reduce the inherent constant time overhead. We implemented the new compilation scheme, which is now the standard compilation scheme used by the JCHR compiler. Empirical evaluation reveals that the new compilation scheme is superior to the traditional one.

The detailed descriptions of the compilation schemes and optimizations presented in this report can readily be used for the compilation of CHR and related rule based languages to any imperative target language. In fact, the new compilation scheme has already successfully been ported to CCHR [WSD07], the C embedding of CHR.

A companion article of this work appears in [VWWSD08]. This report focused on the compilation scheme of JCHR, and gave more details concerning the new compilation scheme and its optimizations. The companion journal article generalizes the compilation schemes presented here to arbitrary imperative target languages, and provides a thorough discussion on other challenges when embedding CHR in an imperative host language.

Bibliography

[AKSS02] Slim Abdennadher, Ekkerhard Krämer, Matthias Saft, and Matthias Schmauß. JACK: A Java Constraint Kit. In M. Hanus, editor, *WFLP '01: Proc. 10th Intl. Workshop on Functional and (Constraint) Logic Programming, Selected Papers*, volume 64 of *ENTCS*, pages 1–17, Kiel, Germany, November 2002. Elsevier. See also `http://pms.ifi.lmu.de/software/jack/`.

[Bau03] Andreas Bauer. Compilation of functional programming languages using GCC—Tail calls. Master's thesis, Institut für Informatik, Technische Univ. München, 2003.

[CF04] John Clements and Matthias Felleisen. A tail-recursive machine with stack inspection. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 26(6):1029–1052, 2004.

[CHR08] The Constraint Handling Rules (CHR) programming language homepage, 2008. http://www.cs.kuleuven.be/∼dtai/projects/CHR.

[CSW03] Wei-Ngan Chin, Martin Sulzmann, and Meng Wang. A type-safe embedding of Constraint Handling Rules into Haskell. Honors thesis, School of Computing, National University of Singapore, 2003.

[DDS07] K. Djelloul, G. J. Duck, and M. Sulzmann, editors. *CHR '07: Proc. 4th Workshop on Constraint Handling Rules*, Porto, Portugal, September 2007.

[DS05] Gregory J. Duck and Tom Schrijvers. Accurate functional dependency analysis for Constraint Handling Rules. In Schrijvers and Frühwirth [SF05], pages 109–124.

[DS07] Leslie De Koninck and Jon Sneyers. Join ordering for Constraint Handling Rules. In Djelloul et al. [DDS07], pages 107–121.

[DSGH03] Gregory J. Duck, Peter J. Stuckey, María García de la Banda, and Christian Holzbaur. Extending arbitrary solvers with Constraint Handling Rules. In *PPDP '03*, pages 79–90, Uppsala, Sweden, 2003. ACM Press.

[DSGH04] Gregory J. Duck, Peter J. Stuckey, María García de la Banda, and Christian Holzbaur. The refined operational semantics of Constraint Handling Rules. In B. Demoen and V. Lifschitz, editors, *ICLP '04*, volume 3132 of *LNCS*, pages 90–104, Saint-Malo, France, September 2004. Springer.

[Duc05] Gregory J. Duck. *Compilation of Constraint Handling Rules*. PhD thesis, University of Melbourne, Australia, December 2005.

[FM04] Th. Frühwirth and M. Meister, editors. *CHR '04: 1st Workshop on Constraint Handling Rules: Selected Contributions*, Ulm, Germany, May 2004.

[Fre08] Free Software Foundation. GCC, the GNU Compiler Collection. `http://gcc.gnu.org/`, 2008.

[Frü98] Thom Frühwirth. Theory and practice of Constraint Handling Rules. *J. Logic Programming, Special Issue on Constraint Logic Programming*, 37(1–3):95–138, 1998.

[Frü05] Thom Frühwirth. Programming with a Chinese horse. Invited Talk at 11th Intl. Conf., CP 2005, Sitges, Spain, October 2005. (slides).

[Frü08] Thom Frühwirth. *Constraint Handling Rules*. Cambridge University Press, 2008. To appear.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.

[HF00] Christian Holzbaur and Thom Frühwirth. A Prolog Constraint Handling Rules compiler and runtime system. volume 14(4) of *Journal of Applied Artificial Intelligence*, pages 369–388. Taylor & Francis, April 2000.

[HGSD05] Christian Holzbaur, María García de la Banda, Peter J. Stuckey, and Gregory J. Duck. Optimizing compilation of Constraint Handling Rules in HAL. volume 5(4–5) of *Theory and Practice of Logic Programming*, pages 503–531. Cambridge University Press, July 2005.

[LY99] Tim Lindholm and Frank Yellin. *The Java(TM) Virtual Machine Specification*. Prentice Hall, second edition, April 1999.

[Pro01] Mark Probst. Proper tail recursion in C. Diplomarbeit, Institute of Computer Languages, Vienna University of Technology, 2001.

[SC$^+$] Vítor Santos Costa et al. YAP Prolog. `http://www.ncc.up.pt/yap/`.

[Sch05a] Tom Schrijvers. *Analyses, optimizations and extensions of Constraint Handling Rules*. PhD thesis, K.U.Leuven, Belgium, June 2005.

[Sch05b] Tom Schrijvers. *The implementation of CHR: A Reconstruction*, 2005. Chapter 5 of [Sch05a].

[SD04] Tom Schrijvers and Bart Demoen. The K.U.Leuven CHR system: Implementation and application. In Frühwirth and Meister [FM04], pages 8–12.

[SF05] T. Schrijvers and Th. Frühwirth, editors. *CHR '05: Proc. 2nd Workshop on Constraint Handling Rules*, K.U.Leuven, Dept. Comp. Sc., Technical report CW 421, Sitges, Spain, 2005.

[SO01] Michel Schinz and Martin Odersky. Tail call elimination on the Java Virtual Machine. In *Proceedings of the ACM SIGPLAN BABEL'01 Workshop on Multi-Language Infrastructure and Interoperability*, volume 59 of *Electronic Notes in Theoretical Computer Science*, pages 155–168. Elsevier, 2001.

[SSD05a] Tom Schrijvers, Peter J. Stuckey, and Gregory J. Duck. Abstract interpretation for Constraint Handling Rules. In P. Barahona and A.P. Felty, editors, *PPDP '05*, pages 218–229, Lisbon, Portugal, July 2005. ACM Press.

[SSD05b] Jon Sneyers, Tom Schrijvers, and Bart Demoen. The computational power and complexity of Constraint Handling Rules. In Schrijvers and Frühwirth [SF05], pages 3–17.

[SSD05c] Jon Sneyers, Tom Schrijvers, and Bart Demoen. Guard and continuation optimization for occurrence representations of CHR. In M. Gabbrielli and G. Gupta, editors, *ICLP '05*, volume 3668 of *LNCS*, pages 83–97, Sitges, Spain, October 2005. Springer.

[SSD06a] Jon Sneyers, Tom Schrijvers, and Bart Demoen. Dijkstra's algorithm with Fibonacci heaps: An executable description in CHR. In M. Fink, H. Tompits, and S. Woltran, editors, *WLP '06: Proc. 20th Workshop on Logic Programming*, T.U.Wien, Austria, INFSYS Research report 1843-06-02, pages 182–191, Vienna, Austria, February 2006.

[SSD06b] Jon Sneyers, Tom Schrijvers, and Bart Demoen. Memory reuse for CHR. In S. Etalle and M. Truszczynski, editors, *ICLP '06*, volume 4079 of *LNCS*, pages 72–86, Seattle, Washington, August 2006. Springer.

[SSW04] Peter J. Stuckey, Martin Sulzmann, and Jeremy Wazny. The Chameleon system. In Frühwirth and Meister [FM04], pages 13–32.

[Sun08] Sun Microsystems, Inc. Java SE HotSpot at a glance. `http://java.sun.com/javase/technologies/hotspot/`, 2008.

[SVWSDK08] Jon Sneyers, Peter Van Weert, Tom Schrijvers, and Leslie De Koninck. As time goes by: Constraint Handling Rules – A survey of CHR research between 1998 and 2007. Submitted to *Journal of Theory and Practice of Logic Programming*, 2008.

[SWD05] Tom Schrijvers, Jan Wielemaker, and Bart Demoen. Poster: Constraint Handling Rules for SWI-Prolog. In A. Wolf, Th. Frühwirth, and M. Meister, editors, *W(C)LP '05: Proc. 19th Workshop on (Constraint) Logic Programming*, volume 2005-01 of *Ulmer Informatik-Berichte*, Universität Ulm, Germany, February 2005.

[TLA92] David Tarditi, Peter Lee, and Anurag Acharya. No assembly required: Compiling standard ML to C. *ACM Letters on Programming Languages and Systems*, 1(2):161–177, June 1992.

[VSD05] Peter Van Weert, Tom Schrijvers, and Bart Demoen. K.U.Leuven JCHR: a user-friendly, flexible and efficient CHR system for Java. In Schrijvers and Frühwirth [SF05], pages 47–62.

[VW05] Peter Van Weert. Constraint programming in Java: een gebruiksvriendelijk, flexibel en efficient CHR-systeem voor Java. Master's thesis, Department of Computer Science, K.U.Leuven, Leuven, Belgium, May 2005.

[VW08a] Peter Van Weert. The K.U.Leuven JCHR System homepage, 2008. `http://www.cs.kuleuven.be/~petervw/JCHR/`.

[VW08b] Peter Van Weert. *K.U.Leuven JCHR User's Manual*, 2008. Available at [VW08a].

[VW08c] Peter Van Weert. Optimization of CHR propagation rules. In *ICLP '08: Proc. 24rd Intl. Conf. Logic Programming*, LNCS, Udine, Italy, December 2008. Accepted.

[VW08d] Peter Van Weert. A tale of histories. In T. Schrijvers, F. Raiser, and T. Frühwirth, editors, *CHR '08*, RISC Report Series 08-10, University of Linz, Austria, pages 79–94, Hagenberg, Austria, July 2008.

[VWWSD08] Peter Van Weert, Pieter Wuille, Tom Schrijvers, and Bart Demoen. CHR for imperative host languages. Submitted to Special Issue of LNAI on Constraint Handling Rules, 2008.

[War80] David H.D. Warren. An improved Prolog implementation which optimizes tail recursion. Research Paper 156, Dept. of Artificial Intelligence, University of Edinburgh, 1980.

[Wie03] Jan Wielemaker. An overview of the swi-prolog programming environment. In *Proc. 13th Intl. Workshop on Logic Programming Environments*, Mumbai, India, 2003. System's home page at `http://www.swi-prolog.org/`.

[Wol01] Armin Wolf. Adaptive constraint handling with CHR in Java. In T. Walsh, editor, *CP '01*, volume 2239 of *LNCS*, pages 256–270, Paphos, Cyprus, 2001. Springer.

[WSD07] Pieter Wuille, Tom Schrijvers, and Bart Demoen. CCHR: the fastest CHR implementation, in C. In Djelloul et al. [DDS07], pages 123–137.

[Wui07] Pieter Wuille. CCHR: de snelste implementatie van CHR. Master's thesis, Department of Computer Science, K.U.Leuven, Leuven, Belgium, May 2007.