K.U.Leuven JCHR User's Manual

Peter Van Weert

Draft Version (December 5, 2006)

Contents

Chapter 1

Introduction

1.1 About the K.U.Leuven JCHR System

The K.U.Leuven JCHR System [VW05, VWSD05, VW06] is a high-performance integration of Constraint Handling Rules (CHR) [SF+06, Frü98] and Java [Sunb]. For proper understanding, we assume the reader familiar with both languages.

As a powerful forward-chaining rule-based language, JCHR allows for very high-level, declarative programming of constraint-based algorithms and applications. The *syntax* is designed to feel familiar to Java programmers, whilst maintaining the declarativeness inherited from CHR. It also allows an easy porting from/to other CHR systems. The intuitive operational *semantics* of CHR [DSdlBH04] – fully adopted by JCHR – remedies the lack of clear semantics sometimes perceived in other, similar rule-based systems. JCHR is currently by far the most efficient implementation of CHR in Java [AKSS01, Wol01, VA06]. Its performance is competitive with state-of-the-art CHR systems in e.g. HAL [dlBDMS02, HdlBSD05] and Prolog [Sch06, SD04, Sch05, HF00].

Disclaimer The current version, version 1.5.0, is a prototype, proof of concept implementation, developed to test and demonstrate the architecture, and to be used as a research vehicle. However, though there might be some implementation issues and bugs distinguishing it from a production-quality system, it is certainly complete and stable enough to be used readily in your applications. The system is also under active development, and each release is a step forward! We kindly ask our users to submit all issues they encounter, so we can keep improving the system.

1.1.1 Compared to other CHR sytems

There exist CHR implementations for several different host languages:

1. **Prolog** After the language's conception in 1991 by Thom Frühwirth, some experimental systems were implemented in, amongst others, ECL^{i}PS^e Prolog [FB95a, FB95b] and Sepia*. The first full implementation of CHR was developed by Christian Holzbaur and Thom Frühwirth for SICStus Prolog [HF99, HF00], and later ported to Yap Prolog. This system is considered the reference implementation. Currently, probably the best, state-of-the-art CHR compiler for Prolog is the K.U.Leuven CHR System [Sch06, SD04, Sch05]. This system was first hosted by hProlog, but soon ported to XSB [SWD03] and SWI Prolog [SWD05]. More recently, the system has also been ported to YAP Prolog, replacing the old reference implementation, and to bProlog [SZD06].

2. **HAL** The SICStus reference implementation was first ported to the HAL language by its original author, Christian Holzbaur. Later the HAL CHR compiler was rewritten by Gregory Duck [dlBDMS02, HdlBSD05].

3. **Haskell** Martin Sulzmann et al. wrote a CHR system for the Haskell variant Chameleon to support their work on customizable type systems [SS01, SS05]. This first Haskell implementation was later replaced by the HaskellCHR implementation [Duc05] by Gregory Duck.

4. **Java** Next to the K.U.Leuven JCHR System, there exist at least three other CHR implementations in Java. More information can be found in the next subsection.

All major CHR implementations (except some of the Java systems, cf. infra) feature similar syntax, often somewhat tailered to the host language, and the same semantics [DSdlBH04]. Of course, this is also the case for the K.U.Leuven JCHR System. Therefore porting programs between these systems is easy, as long as the programs do not rely too much on host language features or built-in constraints. The main differences between K.U.Leuven JCHR and the other systems are indeed mostly due to the greatly different host languages:

- Java is an *imperative* language, and does not offer built-in search/backtracking capabilities. C(L)P languages like Prolog and HAL on the other hand do. This synergy between forward chaining rules and backtracking search is why CHR systems hosted by the latter host languages are very well suited for the high-level declaration of constraint solvers, which is the original goal of CHR. The K.U.Leuven JCHR System can currently only be used to implement purely forward-chaining rule-based programs. This is nevertheless a vast class of programs (in fact, JCHR is Turing complete [tur]). It is possible to implement a search engine on top of Java, as is shown by the systems discussed in the next section, but at the moment K.U.Leuven JCHR comes without such search capabilities.

- Java is an *object-oriented* language. Complex data types (objects) therefore differ greatly from those used in other, more traditional CHR host languages (compound terms). Unlike non-Java CHR systems (and Dynamic Java Constraint Handling Rules (DJCHR)) K.U.Leuven JCHR does not feature symbolic representation and matching of terms. Instead, all data types are Java types (cf. Section 3). We will however see in Section 3.2.4 that it is easy to represent e.g. logical variables as a Java class. Indeed the K.U.Leuven JCHR System is designed to allow arbitrary user-defined types (Section 3) and built-in constraints (Section 4).

- Java is a *strongly-typed* language, whereas the most prominent CHR host language, Prolog, is untyped. We will see in Chapter 2 that this just means JCHR handler and constraint declarations will contain extra type declarations. For the users of the Prolog K.U.Leuven CHR System this should feel very familiar, since type declarations are already commonplace there (as they are indispensible for analysis and optimization).

1.1.2 Compared to other Java CHR sytems

In this section we will discuss three other Java CHR systems, and compare them with K.U.Leuven JCHR:

- Java Constraint Kit (JaCK) [AKSS01, Sch99]

- DJCHR [Wol01]
- chord [VA06]

1.2 About this Manual

This document contains the user's manual for the latest version of the K.U.Leuven JCHR System. At the time of writing, this is version 1.5.0. Documentation for earlier versions of the system will not be maintained (the manual evolves together with the system), so we encourage you to always update to the latest version. This might require some minor changes to source files from time to time, but we think this will never be a real problem: after all, each new version should be a step forward.

1.2.1 Call for comments

The K.U.Leuven JCHR System at the moment is still a proof-of-concept – a concept it is well on its way of proving by the way. Likewise, this manual is only a draft version of what the final version might look like. If something is still not clear after reading this text, please let us know. We are more than happy to help you out, and it will help us improve this manual. If you have any ideas for new features the system should provide, or for other improvements, do not hesitate to let us know. We are always looking for ways to improve the K.U.Leuven JCHR System.

1.2.2 Notation

Sections, paragraphs or sentences annotated like this one (in the margin), concern features or properties only valid for versions of the system higher than or equal to the one in the annotation. Earlier versions will not contain that particular feature or behave differently than described there. **vx.y.z**

FUTURE THOUGHTS

Sections typeset like this one contain general ideas, possible improvements, etc. that might be incorporated in the future. This can give you an idea in which direction the K.U.Leuven JCHR Systemmight be evolving. If there is a problem with the current version (and it is not a bug), there is a big chance it will be in one of these sections. If you do need such a feature, be sure to let us now, and we might be able to accommodate you. We are also always interested in your views on the ideas put forward in these sections.

Chapter 2

Defining a K.U.Leuven JCHR Handler

The K.U.Leuven JCHR syntax is designed to feel familiar to both Java and CHR adapts. Handlers feature strong syntactical resemblance to Java classes, constraint declarations with Java method signatures, etc. Also, the variables and expressions you can use throughout the rule declarations are essentially the same as in Java. The rules themselves on the other hand, try to offer the high-level, declarative approach, inherited from more traditional CHR systems. The latter are mostly implemented in Prolog, but also in other declarative languages, like Haskell and HAL. The proposed language therefore also aims at an easy porting from and to more Prolog-like syntax.

In this chapter, we consider several aspects – mostly syntactical – of a JCHR handler declaration. Like we said before, a handler source file is very similar to a typical Java compilation unit (with a single class declaration):

```
package packagename;

import ...;
...
import ...;

modifiers handler identifier {
    ...
}
```

As illustrated by the above skeleton:

1. The very first thing that has to be done, is to declare to which Java package the handler belongs. More details follow in Section 2.3. **v1.5.0**

2. Next, like we know from a typical Java compilation unit, we import all classes and members we want to use in the remainder of the handler. More information can be found in Section 2.4.

3. Following the (optional) package declaration and imports, comes the actual `handler` block, in which the JCHR constraint handler is declared. This is of course the main part, and will be explained in Sections 2.5–2.8.

4. As in Java, and in fact most programming languages, comments can be used at virtually any place of the code. We discuss comments in Section 2.2.

handler
constraint
solver
option
rules
local
fail

Table 2.1: K.U.Leuven JCHR keywords

Entities	**Allowed initial characters**
handlers, solvers, constraints, rules	Lower case character or '_'
variable, arguments, occurrences	Upper case character or '_'

Table 2.2: Allowed initial characters of a K.U.Leuven JCHR identifier. Note that a dollar mark never is allowed as the first character (because the compiler internally uses this to generate its own, unique identifiers).

First however, Section 2.1 explains which identifiers you are allowed to use in a K.U.Leuven JCHR program. Restrictions regarding identifiers will not be repeated in the remainder of the text, unless this repetition is significant.

FUTURE THOUGHTS

We are considering to take the analogy with a Java compilation unit even further:

- *Ideally, we would like to elevate the* `handler` *declaration to the same level as similar type declarations (*`class`*,* `interface` *or* `enum`*) in a compilation unit. This would mean that multiple types can be declared in the same compilation unit, alongside the handler (even other handlers), as long as there is only one* `public` *type declaration (which does not have to be the handler). This could also mean that inner handler type declarations become possible, etc. Cf. page 12 for related ideas concerning the analogy between handler and class declarations.*

2.1 Identifiers

The identifiers used in a K.U.Leuven JCHR handler to name the different entities (variables, handlers, etc.) have to be valid *Java identifiers*. In short this means that identifiers:

- are composed of alphanumerical, Unicode characters (including `a`–`z`, `A`–`Z`, `0`–`9`), underscore (_) and dollar mark (`$`)
- do not start with a digit
- are not equal to a Java keyword (listed in [GJSB04], Section 3.9) or literal (`true`, `false` or `null`).

For full details on Java identifiers, we refer to Section 3.8 of the Java Language Specification [GJSB04].

We further constrain the identifiers by rejecting the keywords introduced by K.U.Leuven JCHR, listed in Table 2.1. For practical and conventional reasons we also impose some extra limitations on the initial character of the identifiers. Table 2.2 shows an overview.

In the remainder of this text we will no longer come back to these naming rules. We will also never explicitly stress that identifiers have to be unique within their apparent contexts. Identifiers are case sensitive, as in most programming languages.

2.2 Comments

Comments in K.U.Leuven JCHR are completely analogous to comments in Java (cf. [GJSB04], Section 3.7), i.e. there are:

1. Single line comments (a.k.a. *end-of-line comments*): all text from the characters `//` until the end of the line is ignored (including for instance `/*` and `*/`).
2. Multiple line comments (a.k.a. *traditional comments*): all text between the character sequences `/*` and `*/` is ignored (including for instance `//`).

FUTURE THOUGHTS

There exists a third type of comments in Java, JavaDoc comments, similar to a multiple line comment, but enclosed with `/** ... */`. These comments are used for generating API documentation in HTML format. More information can be found on the JavaDoc home page [Sun06c].
Though the generated code is not really intended to be read by users, it would be very interesting not only to deliver compiled handler and constraint classes, but also the accompanying documentation (i.e. JavaDoc). So, it would be nice to allow JavaDoc comments for the handler block, and possibly also for constraint and other declarations, and to propagate them to the generated code. In anticipation, you can of course already write JavaDoc comments: they are simply treated as multiple line comments for the moment.

2.3 The `package` Declaration

v1.5.0

Just like a compilation unit in Java, a JCHR handler source file starts with a package declaration to indicate the package to which the handler belongs[1]. Classes belonging to the same package as the handler will not have to be imported explicitly (cf. infra). Within its package, the handler can also access types and members declared with default access, and class members declared with the *protected* access modifier. From other packages, only *public* types and members are accessible. This is exactly the same as in Java.

If no package declaration is provided, the handler becomes part of the unnamed package. The use of the unnamed package is generally not advised, as classes from the unnamed package cannot be imported, and thus not used, by classes in any other package. It is as [GJSB04] states:

> Unnamed packages are provided by the Java platform principally for convenience when developing small or temporary applications or when just beginning development.

Although not yet enforced by the K.U.Leuven JCHR compiler, it is the convention to store the JCHR handler source files in the same directory as used for the package. Following another convention – naming the handler source file after the JCHR handler it declares – will then ensure that there can never be more than one handler with the same name in the same package.

FUTURE THOUGHTS

- *We might start enforcing (some of) the above conventions soon.*
- *Accessing non-public members is currently not yet supported, even if they are part of the same package as the handler. This will be remedied in one of the next releases.*

[1] It is in fact the package to which the generated handler class file belongs: cf. Section 5.2.2.

2.4 `import` Declarations

The `import` statements used by the K.U.Leuven JCHR are completely equivalent to the ones we know from Java. This means they are an *optional* tool allowing a programmer to tell the compiler which Java types (classes or interfaces) might be used by the program. Without the use of an appropriate import declaration, the only way to refer to a type declared in another package, or a static member of another type, is to use its cumbersome *fully qualified name* (cf. infra).

In the remainder of the section we give an overview of all essential aspects of the four different import declarations supported by the K.U.Leuven JCHR System. We always adhere to the Java Language Specification [GJSB04] as much as possible. For example: classes in the package `java.lang` do not have to be imported explicitly, nor do classes that are part of the same package as the handler (cf. the previous section). **v1.5.0**

FUTURE THOUGHTS

- *There are some minor details that are not yet implemented: the compiler will not complain when importing non-accessible types or members, importing a type with the same simple name as the handler class or a constraint class, etc. These minutiae should never pose any problems though.*

2.4.1 Single type imports

The syntax for a *single-type import* is:

```
import fqn;
```

with *fqn* a *fully qualified name* of a Java class or interface. For example:

```
import java.util.List;
```

The *fully qualified name* of a class or interface includes its package name (`java.util` in the above example) followed by the *simple name* of the type itself (`List`). Once imported, you can (but you do not have to) use a type's simple name (i.e. without the package name) instead of the fully qualified name in the rest of the code.

It is not possible to import two types with the same simple name. For example, the JCHR compiler will reject:

```
import java.util.List;
import java.awt.List;
```

The best you can do here is importing only *one* of the conflicting classes (e.g. the one that is used the most in the handler) and use the fully qualified name for the remaining class(es). Implicitly imported types (that is: classes in `java.lang`, or in the package the current handler is declarad part of) will never conflict with other imports. Next section explains the same holds for types imported 'on demand'. **v1.5.0**

All this (and more) is indeed *completely* analogous to single type imports in Java. More detailed information can be found in Section 7.5.1 of [GJSB04].

2.4.2 Type imports on demand

v1.3.2

type-import-on-demand declarations, as defined in Section 7.5.2 of [GJSB04], are also supported by the K.U.Leuven JCHR System. A type-import-on-demand declaration imports all the types of a named type or package as needed. For example

```
import java.util.*;
```

allows you to address all types in the `java.util` package using their simple name. Unlike in Java it is not a compile-time error for a type-import-on-demand declaration to name a type or package that is not accessible or does not exist. Other than that, type-import-on-demands behave completely analogously to their Java counterpart. This means e.g. that single-type imports *shadow* type-imports-on-demand. For example, if your import section looks like this:

```
import java.awt.*;
import java.util.List;
```

the simple name `List` will refer to `java.util.List`, and not to `java.awt.List`. Although it is perfectly correct to do the following imports:

```
import java.awt.*;
import java.util.*;
```

using the simple name `List` will then result in an ambiguity exception. For more information, we refer to the corresponding section of [GJSB04].

2.4.3 Single static imports

v1.4.0

A *single-static-import* declaration, as introduced to the Java language since version 5 (all details can be found in [GJSB04], Section 7.5.3), imports all static members with a given simple name from a type. Static members can be static fields (e.g. `java.lang.Math.PI`), static methods (e.g. `java.lang.Math.sqrt`), or a static inner class (e.g. java.util.Map.Entry). The syntax looks like:

```
import static fqn.simple name;
```

If you e.g. use the following import declarations

```
import static java.lang.Math.PI;
import static java.lang.Math.sqrt;
```

you can calculate the radius of a circle with given area `a` as follows:

```
sqrt(a / PI);
```

whereas without static imports you had to use the following, more cumbersome notation:

```
Math.sqrt(a / Math.PI);
```

2.4.4 Static imports on demand

v1.4.0

A *static-import-on-demand* declaration is to a single-static-import what an import-on-demand declaration is to a single-type-import. For example:

```
import static java.lang.Math.*;
```

allows you to address all (static) constants and methods from the `Math` class using their simple names. For all details regarding these import declarations we refer to the Java Language Specification [GJSB04], Section 7.5.4.

Tip: In anticipation of parser support for arithmetic and boolean logic expressions, you can use static imports to import the methods of `util.ArithmeticsUtils` and `util.BooleanUtils`. **v1.4.0**

Tip: Static imports also come in handy when using enumeration types: you can statically import `enum` values using single-static-imports, or you can directly import all values of an `enum` type using a static-import-on-demand.

2.5 The `handler` Declaration

The first thing you have to do is declarating the name of the constraint handler[2]. This is the same in a typical Prolog CHR system[3]. To complete the syntactical analogy with a Java class declaration (cf. previous sections), a K.U.Leuven JCHR handler is a Java code-block (i.e. enclosed with "{" and "}"). The full skeleton of a typical handler source file looks like:

```
package packagename;

import ...;
...
import ...;

modifiers handler identifier {
    ...
    /* options and declarations */
    ...
    rules {
       ...
    }
}
```

v1.5.0

The handler declaration will most often be preceded by a `public` *access modifier*. A `public` handler can be used by any other class. Handlers with no access modifier are said to have *default access*, and can only be accessed by classes within the same package. This is analogous to what is done in Java[4].

The handler block itself currently consists of:

1. Declarations of user-defined constraints and built-in constraint solvers. These are further explained in Section 2.6. For the constraint declarations, the main conceptual difference with many other (non-Java) CHR implementations is that argument-types are mandatory.

2. Compiler options, as described in Section 2.7. Declarations and options can be freely interleaved , as long as they all precede the `rules` code-block. **v1.4.0**

3. A single, mandatory `rules` block (Section 2.8) containing the core of a JCHR handler: the JCHR rules. The syntax K.U.Leuven JCHR uses for the rules is very close to the one used in the literature [Frü98], and by most other CHR systems [H$^+$06, SD04, HdlBSD05]. The main difference is that Java expressions and statements can be used in the rule declarations.

Porting a *pure* CHR handler written for another CHR system is thus simply a mather of:

1. Adding correct (Java) types for all arguments in user-defined constraint declarations.

[2] Actually, in JCHR, the first thing that is done are the package declaration, followed by a series of import declarations. We however do not really consider them part of the handler declaration: they are just part of the enclosing Java compilation unit of which the handler is one type declaration (be it currently the only allowed declaration).

[3] At the time of writing, the `handler` declaration of the K.U.Leuven CHR System – only introduced for compatibility with the old SICStus reference implementation – is deprecated and will be discontinued in the near future. It is very unlikely this will ever happen in K.U.Leuven JCHR, since the name of handler is used for the name of the generated class.

[4] `private` or `protected` are not valid as access modifier of a handler. The former because the handler declaration currently is the only allowed type declaration of the file, the latter because a handler declaration is always a top-level type declaration.

```
import runtime.*;

handler leq<T> {
        solver EqualitySolver<T>;

        constraint leq(Logical<T>, Logical<T>) infix =<;

        rules {
                reflexivity  @ X =< X <=> true.
                antisymmetry @ X =< Y, Y =< X <=> X = Y.
                idempotence  @ X =< Y \ X =< Y <=> true.
                transitivity @ X =< Y , Y =< Z ==> X =< Z.
        }
}
```

Listing 1: A generic, lesser-than-or-equal handler in K.U.Leuven JCHR

2. Copying the rules.
3. Adjusting the host language statements (arithmetics, ...) in guards and bodies to the corresponding Java statements.

A *pure* CHR program does not use features of the host-languange, which are not readily available in Java. In the context of porting from Prolog, this includes the use of disjunction/backtracking in rule bodies, or symbolic matching on compound terms. If either of these is used in an essential way, porting becomes more challenging. This is however not surprising, because these are not features of CHR, but of the host language. We already pointed out these differences in Section 1.1.1.

FUTURE THOUGHTS

Other parts might be added to a handler in the future:

- *To strengthen the analogy between a class and a handler, it would be very interesting to allow a handler to have* members *(fields, methods, inner classes, etc.).*
- *We could add* goal sections*, similar to the ones used by JaCK [Sch99, AKSS01]. For some more information and related ideas, cf. page 41.*
- *Another interesting feature would be to allow the user to specify queries in a declarative manner. Resulting methods would then (efficiently) return a collection of all constraints satisfying user-defined conditions specified in the query declarations.*

2.5.1 Generic handlers

To be able to declare *polymorphic* handlers K.U.Leuven JCHR introduced the notion of *generic handlers*. An archetypical example of a polymorphic (generic) handler is `leq`, listed in Listing 1. Without support for generics a distinct solver would have to be declared here for each variable type (handlers like integer_leq, string_leq, list_leq, ...). Alternatively, one could write one general handler (like object_leq), but this would necessitate explicit type casting afterwards. Clearly, generic handlers are a much more powerful, legible and robust solution. They also ease the transition from traditional, typically untyped CHR systems to strongly typed K.U.Leuven JCHR.

Generic handlers are, of course, completely analogous to generic classes in Java. Since *generics* in Java is a relatively new concept[5], we will elaborate somewhat

[5] At least at the time of writing...

further. More information on generic types in Java can be found in [Bra04] and in [GJSB04].

Generics extends the handler declaration with a list of *(formal) type parameters*, enclosed by < and >. When a generic handler is instantiated, these will be bound to *actual type arguments* (cf. Section **??**). These will be a *reference type* – i.e. they cannot be a primitive type.

Subsequent type parameters are separated by commas. Each type parameter can have several *upper bounds*, limiting the actual type arguments that can be used. An upper bound is a reference type or one of the type parameters *different from and left of* the current parameter.

```
handler a_handler<T extends Number, S extends T> {...}
```

An upper bound can itself be a parameterized type. Using formal type parameters is allowed in a parameterized upper bound, as long as it is located to the left of the place of use (i.e. *including* the parameter you are bounding). Several upper bounds can be used, separated by &'s.

```
handler some_handler<T extends Cloneable & Comparable<T>> {...}
```

Since Java does not support multiple inheritance, *only one* class is allowed as upper bound. Moreover, this then has to be the first in the list. Each subsequent upper bound will have to be an interface. When using a type parameter as upper bound, no other upper bound is allowed.

```
handler my_solver<T extends Number & Cloneable, S extends T> {...}
```

Tip: The name of a type parameter can in principle be equal to any Java simple name, but to avoid confusion[6] there exist some conventions [Bra04]:

> We recommend that you use pithy (single character if possible) yet evocative names for formal type parameters. It's best to avoid lower case characters in those names, making it easy to distinguish formal type parameters from ordinary classes and interfaces. Many [...] use E, for element, [...] K for keys and V for values. [...] We use T for type, whenever there is not anything more specific about the type to distinguish it. [...] If there are multiple type parameters, we might use letters [...] such as S.

FUTURE THOUGHTS

Wildcards are not yet supported by the K.U.Leuven JCHR System, nor are lower bounds (see [Bra04] if you do not know what these are). This means for example that you cannot yet write

```
handler mergesort<T extends Comparable<? super T>>
```

(both extra features would be needed here). Luckily, even though the above typing is certainly better, for most practical uses the following suffices:

```
handler mergesort<T extends Comparable<T>>
```

2.6 Declarations

Before you can define the actual JCHR rules (cf. Section 2.8), you have to declare which user-defined constraints you will be handling (Section 2.6.1), and which built-in solvers the compiler can use to solve built-in constraints (Section 2.6.2). The

order in which the different declarations are done is free , as long as all declarations precede the `rules` code-block. The declarations can also be interleaved freely with compiler options (Section 2.7). **v1.4.0**

2.6.1 `constraint` declarations

As in other CHR implementations the first thing to do, after the handler declaration, is declaring which constraints you will be defining in the remainder of the handler. Constraints are declared using the `constraint` keyword, followed by a comma-separated list of constraint declarations. **v1.5.0**

Because K.U.Leuven JCHR is a statically typed CHR dialect, each constraint declaration will include, besides its unique identifier, the formal type of each of its parameters. In principle, any legal Java type is allowed, including primitive types, generic types, type parameters, etc. Detailed information on type restrictions is provided in Section 3.

A constraint declaration also includes an *access modifier* . As always in Java-like languages, this modifier precedes the rest of the declaration. Valid access modifiers are `public`, `protected`, `private` and `local`, the latter being the only non-Java keyword. When no access modifier is provided, we say the constraint has *default* access. Next we will give some intuitions to what these modifiers mean. The concrete implications will probably become clearer when looking at the actual generated code, and the access modifiers therein. Some more details on the consequences of access modifiers can be found in Section 6.3. **v1.5.0**

- `public` constraints are constraints that, in the generated code, are accessible to any other class that can access the handler class (cf. supra). These constraints can thus be told (that is, more or less, added to the store) from outside the handler, the content of their constraint store can be inspected and queried, etc.

- The most restricted access modifier is `private`, used to declare constraints that are invisible from outside the handler. These constraints can thus only be told in the body of rules, the constraint store can never be inspected, etc. This access modifier is most often used for intermediate or auxiliary constraints, or for constraints that should only be told under controlled conditions (like assertion order, presence of other constraints, ...). A final advantage of `private` constraints is that, as they are virtually invisible, they can easily be changed, without affecting any user code. Also, the compiler has a bit more freedom when compiling these constraints, which could result in performance improvements.

- If a constraint is declared to have default access (i.e. no access modifier was specified), only classes within the same package as the handler can tell and inspect the constraint. These are constraints you want to use in trusted classes (and thus put in the same package), typically written by yourself, but without exposing them to classes from other users, possibly because uncontrolled use of these constraints can result in unwanted results. **Tip:** To protect the entire handler from unauthorized access, you can also use a non-public access modifier for the handler (cf. Section 2.5). Then, even if it has declared `public` constraints, code from outside the package can never access any of its constraints, as they cannot access the enclosing handler class.

[6] As in Java it is for example allowed to give a type parameter the same name as a class or interface. `handler leq<Integer> {...}` for example is perfectly valid. These parameters always have precedence over other identifiers, leading to unwanted confusion.

- The final access modifier known from Java, `protected`, is similar to default access, only that subclasses of the generated handler class can also access the generated constraint code (even if they are part of a different package).
- There is one last, often occurring type of constraints that cannot yet be declared with the above, standard Java access modifiers. For this, we introduce an extra, JCHR specific access modifier: `local`. `local` constraints are constraints that can only be told from within the JCHR rules, but who's constraint stores can nevertheless be inspected by external code. A `local` constraint is thus a constraint that is half `private` (the telling part), half `public` (the constraint store is publicly inspectable). This mode of access is particulary useful for constraints that represent results for some JCHR program.

A constraint declaration is thus syntacticly very similar to a method signature in Java (access modifier, unique identifier, followed by a comma-separated list of argument types, enclosed by round parenthesis). To strengthen this analogy even further you can also name each of the parameters. **Tip:** You are strongly advised to name the arguments of a constraint, because it renders the generated code more legible and usable.

FUTURE THOUGHTS

At the moment, if the user does not provide an explicit identifier for a certain argument, the compiler assigns a unique identifier – `$i` if the i'th argument is unnamed. This results in getter methods of the form `get$i()` in the generated constraint class. We might make the latter inspector methods unaccessible in the future. Or maybe a more fine-grained way to specify the accessibility of these getter methods is warranted?

The default notation for constraints is the familiar *prefix* notation (cf. Section 2.8.3), but binary constraints can also be written *infix*. Some constraints simply read better that way. An example can be found in the `leq`-handler in Listing 1. To declare that a constraint can be written infix, the constraint declaration has to be followed by the `infix` keyword and a (unique) *infix identifier*, or even a comma-separated list of infix identifiers. Valid infix identifiers are: **v1.5.0**

1. The ten *built-in infix identifiers*: `=`, `==`, `===`, <, >, `<=`, `=<`, `>=`, `!=` and `!==`. **v1.5.0**
2. *User-defined infix identifiers*: arbitrary[7] unicode character sequences surrounded with apostrophes (single quotes), e.g. `'∼>'`, `'is larger than or equal to'` or `'<>'`. **v1.5.0**

2.6.2 `solver` declarations

Because in principle arbitrary built-in solvers can be used, the concrete built-in solvers the JCHR compiler can utilize to compile the high-level built-in constraints in a handler have to be declared. You can read more about this feature, and the solvers known implicitly by the compiler, in Section 4. Declaring a solver is done using the `solver` keyword, followed by the class or interface of the built-in constraint solver. As always, these types can be parameterized. It is also possible to assign a solver a unique identifier in the usual way. In subsection 2.8.3 on page 18 we will see why this can be useful to disambiguate constraint identifiers.

[7] You can even use escaped characters (even escaped single quotes!) if you feel the need.

2.7 Compiler Options

Like most more mature systems the K.U.Leuven JCHR offers the possibility to pass options down to the compiler. The syntax is:

```
option(option name, value);
```

Supported are in principle options with boolean, integer or string literals as value. Certain special purpose option values have also been added for particular options. Following Prolog `on/off` is also supported as boolean literals for options, next to the more common Java-like `true/false`. For completeness, `yes/no` can also be **v1.4.0** used. An overview of the options that can currently be set in JCHR handler source code: **v1.4.0**

Name	Values	Default	Description
`hash`	boolean	`true`	Toggles use of hash indices
`debug`	`off/default/full`	`default`	See Section 7.1

Certain options (currently all the above) can also be set by providing it to the compiler using command line options, as we will see in Section 5.2.1. The latter options will override options set in the source code if they have the same name.

2.8 The `rules` Section

The `rules` code block is the very heart of a JCHR handler declaration. Before we come to the most essential part however, the declaration of the actual JCHR rules, we first need to say a few words on variables and typing.

2.8.1 Variable declarations

Since K.U.Leuven JCHR is strictly typed, the compiler needs to know the type of *each* variable. Luckily, for variables used already in the head of the rule, the compiler can infer the type from the declaration of the constraint. So variables that are used **v1.4.0** as a top-level argument in a head of a rule do not have to be declared explicitly. For more information on these *implicit variable declarations* in rule heads, we refer to Section 2.8.4.

Variables that are used for the first time in the body of a rule, referred to as *local variables*, do have to be declared *explicitly*. The `variable` keyword, which was borrowed from JaCK [Sch99, AKSS01], has been deprecated. Instead, to declare **v1.4.0** the type of a *local* variable, the `local` keyword is used, followed by the type of the variable and a comma-separated list of identifiers. These identifiers cannot start with an underscore, as these identifiers are reserved for anonymous variables (cf. **v1.3.0** Section 2.8.4). Rule declarations and variable declarations can be interleaved freely, **v1.4.0** as long as the declaration of local variables is done before it is used for the first time. If you declare a variable, but never use it in the body of some rule, a warning is raised. **v1.4.0**

A non-local variable declared implicitly in the head of some rule, with the same identifier as used earlier in a local variable declaration, will hide the local variable. **v1.4.0** It is then also no problem if their types do not match. If the identifier of a local variable is equal to the simple name of some (imported) reference type, or the name of a (statically imported) field, a warning is given by the compiler because the **v1.4.0** variable will hide the type or field, which could lead to unwanted confusion.

On a more semantic level: the variables you are declaring in K.U.Leuven JCHR are *not* logical variables per se as is the case in JaCK [Sch99, AKSS01]; instead, these variables are true Java variables, with actual Java types. More information on type restrictions can be found in Section 3.

FUTURE THOUGHTS

- *The fact that local variables have to be declared before (i.e. higher in the source code) is, however logical, only an implementation issue. This restriction might be relaxed in later versions.*
- *You cannot yet use a variable for the first time in the guard of a rule. There is no real reason for this, it is just not yet implemented.*
- *There is a small implementation issue with first-time uses of variables in non-variable expressions in the head of a rule: cf. the future thoughts paragraph in Section 2.8.4.*

2.8.2 Rule Structure

A rule consists of three major parts, each described in detail in one of the subsequent sections: the *head* (Section 2.8.4), the *guard* (Section 2.8.5), and the *body* (Section 2.8.6).

As in most, if not all, CHR systems, there are three types of rules. The basic syntax for a *propagation rule* is:

```
head ==> guard | body
```

Similarly, for a *simplification rule*:

```
head <=> guard | body
```

And finally a *simpagation rule* looks like:

```
kept \ removed <=> guard | body
```

An empty guard (`true`) can be omitted, together with the '`|`' symbol. Heads can never be empty. Optionally, you can name each rule by prepending it with a unique identifier followed by an '`@`' symbol. This name has no semantical meaning. It functions as mainly documentation, but will also make compiler output – e.g. errors and warnings – and tracing more legible.

Separators

All parts of JCHR rules are essentially conjunctions. In the next sections we will see which types of conjuncts are supported and allowed in each part of a rule. Since this is a port of CHR to *Java*, conjuncts are separated by double ampersands ('`&&`'). To indicate the end of a rule, another typical Java separator is used: the semicolon ('`;`'). To accommodate easy porting from existing systems, more Prolog-like syntax is also supported, meaning colons (',') and a dot ('.') respectively.

FUTURE THOUGHTS

- *The dot at the end of a rule is somewhat troublesome, because it is used in the Java language for other purposes. One problem in the current version is e.g. that you cannot end a rule with a number followed by a dot, because this is interpreted as part of the number by the parser.*
- *There are some other syntactical variants that are worth considering. For example, we could use a more Java-like syntax for guards:*

```
head ==> if (guard) body
```

2.8.3 Conjuncts and arguments

Regarding the conjuncts and arguments that are allowed, the K.U.Leuven JCHR System tries to remain as close as possible to the host language Java (this is not the case in other embeddings of CHR in Java [AKSS01, Wol01, VA06]). We know already from Section 2.8.1 that the variables used in K.U.Leuven JCHR are in the first place the same as the variables we know from Java, i.e. not some form of (typed) logical variables. In the remainder of this section, you will see that we followed a similar philosophy for the other expressions in the JCHR language.

Conjuncts

Constraints The first type of conjuncts is of course the *constraint*. As always in a CHR system, there are two kinds of constraints: *user-defined constraints*, declared by the enclosing JCHR handler, and *built-in constraints*, solved by a built-in constraint solver, orthogonal to the JCHR core system.

User-defined constraints are written using a familiar *prefix* notation, namely

constraint`(`$\texttt{arg}_1$`, ..., `$\texttt{arg}_n$`)`

where *constraint* is a n-ary user-defined constraint, declared by the enclosing handler (cf. Section 2.6.1). *Binary constraints* for which an infix identifier is declared (as explained in Section 2.6.1), can however also be written using an *infix* notation. For *nullary constraints* (also called *flag constraints*), the '`()`' can be omitted . **v1.4.0**

Built-in constraints are syntactically completely analogous to user-defined constraints (cf. supra). They are solved by one of the declared or implicitly known *built-in constraint solvers*. Binary built-in constraints can also have infix identifiers. All details can be found in Section 4. There are also two special trivial built-in constraints that can be used:

- `true`: constraint that always succeeds
- `false` or `fail`: (nullary) constraint that always fail. `fail` also has a variant that takes as single argument a `String`, which can be used to indicate the reason of failure. **v1.5.0**

These built-ins are mainly intended to be used in bodies: they can be used in guards as well, but in general one omits trivial guards.

Name clashes If a built-in constraint has the same identifier (prefix or infix) and arity as a user-defined constraint, then the user-defined constraint will get precedence over the built-in one[8]. We saw however in Section 2.6.2 how you can assign an identifier to a built-in solver. If a built-in constraint's name clashes with that of a user-defined constraint, you can precede its identifier with that of its built-in constraint solver: for prefix notation this becomes *solver*`.`*constraint*`(...)`; for infix notation `...` ´*solver*`.`*infixid*´ `...`. You can also use the special keyword `$builtin`, instead of a solver identifier, to indicate that the intended constraint is built-in and not user-defined (the compiler will then try to infer the built-in constraint solver). Of course, solver identifiers can also be used to disambiguate between two or more built-in constraints (declared by different constraint solvers) that have the same identifier and arity.

[8] Name clashes are in fact only a real problem if both constraints have the same or similar parameter types: the compiler will try to disambiguate based on the typing information about the arguments.

Host language statements A second type of conjuncts are *host language statements*. This is the same for all CHR systems. The main difference here is that Java statements are further away from constraints than e.g. Prolog statements. In principle, any Java statement could be a valid conjunct, but the current parser and compiler only support the following subset: statements that can be used as a conjunct are:

- method invocations
- constructor invocations **v1.4.0**
- field accesses (boolean fields in guards only)
- variables (boolean variables in guards only) **v1.4.0**

Arguments

The following expressions can currently be used as argument for a constraint, method invocation or constructor invocation:

- method invocations (if they do not have `void` as return-type)
- field accesses
- constructor invocations
- variables
- numeric literals (i.e. integer or floating point literals, completely compliant with the Java Language Specification [GJSB04])
- character literals (a character or an escape sequence, enclosed in ASCII single quotes, as defined in [GJSB04].)
- string literals (zero or more characters enclosed in double quotes: again, completely analogous to [GJSB04])
- boolean literals (`true` or `false`)
- the `null` literal

For character and string literals identical escape sequences as defined in [GJSB04] are supported.

Implicit arguments An implicit argument is that part of a method invocation or field access preceding the final dot. Are supported as implicit argument:

- field accesses
- class names (i.e. for static members)
- variables

Note that e.g. method invocations cannot yet be used as an implicit argument for e.g. another method invocation.

FUTURE THOUGHTS

The supported Java expressions are currently quite limited. Though all most commonly used expressions are already possible, it would be nice to really allow arbitrary Java expressions and statements in rule definitions. Parsing and modelling the entire Java language is however not an easy task.

2.8.4 Heads

A *head* is a non-empty conjunction of user-defined constraints. In a simpagation rule the head is split in two non-empty parts by a backslash. A conjunct of a head is also called an *occurrence* of that particular user-defined constraint.

Implicit variable declarations

v1.4.0

If you use a variable in the head of rule, you do not have to declare it explicitly like you have to do with `local` variables (cf. Section 2.8.1). The reason is that the compiler can infer the variable type from the type information given in the `constraint` declarations (Section 2.6.1). If you use the same variable more than once in the head (cf. also below under implicit guards), each of these variable occurrences has to have the same type. If used in different rules on the other hand, the same variable identifier can be used to identify variables that are typed differently.

FUTURE THOUGHTS

We could relax the above restriction in the future, but it is doubtful this would be worth the effort. Using the same identifier in a single rule for differently typed variables (or one multi-typed variable) would mean it is no longer always clear in the guard and the body which type a variable has exactly, which would lead to more messy ambiguities. It is also hardly ever needed, and you can always use two different variable names and an explicit equality guard in the exceptional cases where it is needed.
One idea would to allow different types, but only if the variables are not used in any guard or the body. This way only an implicit equality guard would have to be created between two differently typed variables, which is no problem.

As we said before, if a variable is declared implicitly in the head of a rule, it will hide a local variable if it has the same identifier (i.e. they are allowed to have different types). If a variables has the same identifier as an (imported) reference type, it will hide that type name. A *warning* is generated by the compiler in that case, because this leads to certain pathological cases like:

```
constraint c(Integer);
...
c(String) ==> foo(String.valueOf(13)).
```

In the above example, the semantics dictates that `String` in the body is a variable of type `Integer`, and thus that `String.valueOf(13)` is of type `Integer`, whereas most probably the user intended it to be of type `String`, the return type of the static `valueOf` method of class `String`.

Also, (statically imported) fields whose name start with an upper case letter could be mistaken for an implicit declaration of a variable if used as a top-level argument in a head. The Java code conventions [Suna, GJSB04] however state that only constants should start with an upper case (in fact: constants should have all upper case names). Therefore the compiler will only consider a capitalized name an implicit variable declaration if it is not a constant (i.e. annotated with `final`) field. Using non-final capitalized fields in the head will require the explicit use of the implicit argument (so you cannot profit from static imports there), but this should never occur if the naming conventions are followed. Note that this semantics directly allows the use of statically imported `enum` values in the head of rules, since, under the hood, these are implemented in Java using static final fields.

We will see below that besides variables, other (host language) expressions can be used as an argument in rule heads. Only *top-level* uses of variables count as *implicit variable declarations* (other uses will, as seen below, result in implicit guards). If you never use a variable as a top-level argument, and you use it in elsewhere in a

head or a guard, a compiler error will be raised as the variable is not declared. For use of variables in bodies only (cf. infra), we know already you can use *local* variable declarations (Section 2.8.1).

FUTURE THOUGHTS

There are some issues with the current implementation if you use a variable for the first time in a non-variable expression. For example, the following rule uses the variable `X` *two times before it is used as a top-level argument:*

```
a(X.increment()), b(Math.abs(X), X) <=> true.
```

Note that if `X` *would not have been used top-level in the third argument, the rule would have been invalid. To overcome this small implementation glitch, you can:*

- *Rearrange the heads such that one of the top-level uses becomes the first use. This might not be possible if the rule in question is a simpagation rule or, as in our example, if non-top-level variable uses occur within the same constraint as its top-level occurrence.*
- *Write the implicit guards explicitly: i.e. introduce a new variable and move the non-variable expression to an explicit equality guard. For example, a combination of the first two techniques can be used to rewrite the above rule to:*

  ```
  b(Y, X), a(X.increment()) <=> Y == Math.abs(X) | true.
  ```

- *A third, pragmatic ad-hoc solution has been added as well, that does not require you to change your rules: you can declare the variable using a* `pragma_var` *declaration directly preceding the rule. These declarations are similar to local variable declarations, but are only valid in the rule directly after it (two or more* `pragma_var` *declarations can be used for the same rule):*

  ```
  pragma_var MyFabulousType X;

  a(X.increment()), b(Math.abs(X), X) <=> true.
  ```

  This type of declaration will be, first deprecated and ignored, and later discontinued as soon as the implementation deals with non-top-level variables properly. You can then simply delete these temporary declarations.

Implicit guards

Using the same variable more than once as an argument results in what is called an *implicit guard*. For example:

```
c(X, X), d(Y), c(X, Y) ==> ...
```

will be rewritten internally by the compiler to a form similar to[9]:

```
c(X, X1), d(Y), c(X2, Y1) ==> X == X1 && X == X2 && Y == Y1 | ...
```

This means of course that, if you use variables more than once in a head, there must exist a suited built-in to test the equality of two variables of their type. For more information on equality built-in constraints, we refer to Section 4.4.

You can also use non-variables as argument to a constraint in a head (cf. Section 2.8.3 for supported arguments). This also results in an implicit guard. For example:

```
c(0, Integer.parseInt(X)) ==> ...
```

will be rewritten to:

[9] The concrete normal form actually depends on the occurrence under consideration, and on the join ordering chosen.

```
c(X0, X1) ==> X0 == 0 && X1 == Integer.parseInt(X) | ...
```

A non-variable expression occurring in a head does not have to be of the exact same type as the formal type at its argument position; there does however have to exist a built-in equality constraint capable of testing whether the expression used is equal to an expression of that formal type (cf. Section 4.4).

Implicit guards are nothing more than syntactic sugar, that makes it easier do limit the constraints that matched by a rule.

Anonymous variables

v1.3.0

If you use a variable as an argument in the head of a rule, but never again in the body or an (implicit or explicit) guard, a warning will be raised by the compiler, **v1.4.0** because this could indicate a programming mistake. This type of unused variables is commonly called *singleton variables*. To prevent these warnings from being generated, you can use *anonymous variables*.

An *anonymous variable* is a variable whose name starts with an *underscore* ('_'), and can only be used in the head of a rule. There are in fact two kinds of anonymous variables:

1. The nameless anonymous variable, whose identifier is simply _. This identifier can only be used as a top-level argument in the head. Using it more than once will never result in an implicit guard. Its different occurrences also do not have to be of the same type (each occurrence is considered a different variable).

2. Named anonymous variables – note the contradiction in terminis! – have identifiers that start with but are not equal an underscore. The first non-underscore character can be any character valid for an identifier (cf. Section 2.1), i.e. it is not limited to a capital letter. This type of anonymous variables can be used to better document your code (when compared to the nameless one), whilst still preventing obsolete warnings. Also, using the same named anonymous variable more than once *will* result in an implicit guard. **v1.4.0** Consequently, each occurrence of an anonymous variable in the same rule will be required to have the same type.

It is important not to forget that anonymous variables, also the named ones, can only be used as top-level arguments of a rule head (i.e. not even in an implicit guard).

2.8.5 Guards

A guard can only contain:

1. Built-in constraints that can be asked. For more information on ask and tell versions of built-in constraints see Section 4.1.

2. Host language expressions of type `boolean` or `Boolean`, or that can be coerced to a boolean type. Cf. Section 2.8.3 for more information on supported Java expressions.

In a guard only variables are allowed that have been used in the head of its rule.

FUTURE THOUGHTS

There is no valid reason for this limitation: it is just not implemented yet. This should be dealt with in one of the next versions.

2.8.6 Bodies

The body of a rule can contain built-in constraints that can be told (cf. Section 4.1), user-defined constraints, and any Java statement supported (Section 2.8.3). In a body local variables can be used – that is, variables that do not yet occur in the head if the rule – provided they are declared in advance by a `local` declaration, as seen in Section 2.8.1.

Tip: Only user-defined constraints declared in the enclosing handler are considered user-defined. It is however also possible treat constraints of other K.U.Leuven JCHR handlers as built-in tell constraints (i.e. use them in the body of a rule) if you declare their handler as a built-in constraint solver (cf. Section 4.5). **v1.2.0**

2.8.7 Pragmas

Rules can be annotated with pragmas. In general, a *pragma* is defined in computer science as:

> a compiler directive embedded in source code by programmers, communicating additional "pragmatic" information (on control, optimizations, ...)

A comma-separated list of pragmas can be added at the end of a rule, preceded by the `pragma` keyword. Some pragmas can also be declared *in-head* (cf. infra). **v1.4.0**

For use in pragmas, occurrences can be given an identifier by adding a '`#`' followed by a unique identifier directly after it.

pragma passive [10] This is probably the best-know pragma for CHR, but nonetheless best left to more advanced users. If you declare an occurrence passive, no code will be generated for that particular occurrence. This means that if a constraint becomes active, it will not see that occurrence, it is passive in that occurrence. In other words: that occurrence will only be used to match constraints, never to seed a search for partner constraints. If you are familiar with the working of the Rete algorithm [For82], this is similar to a join-node that cannot be right-activated (CHR does not keep a beta-network though, as it uses an algorithm closer to LEAPS [MBLG90]). This changes the behavior of the CHR system, because normally, a rule can be entered starting from each occurrence. Usually this pragma will improve the efficiency of the constraint handler, but care has to be taken in order not to lose completeness.

For example, in the handler `leq` (listed on page 12), any pair of constraints, say `A <= B`, `B <= A`, that matches the head `X <= Y`, `Y <= X` of the `antisymmetry` rule, will also match it when the constraints are exchanged, `B <= A`, `A <= B`. Therefore it is enough if a currently active constraint enters this rule in the first head only, the second head can be declared to be passive. Similarly for the `idempotence` rule. For the latter rule, it is more efficient to declare the first head passive, so that the currently active constraint will be removed when the rule fires (instead of removing the older constraint and redoing all the propagation with the currently active constraint).

Declaring the first head of rule `transitivity` passive would however change the behavior of the handler. It will propagate less depending on the order in which the constraints arrive (we use a Prolog-like notation here to illustrate):

```
| ?- X =< Y, Y =< Z.
X =< Y,
Y =< Z,
```

[10] Parts of the explanation in this paragraph is adapted from [H+06].

```
import runtime.*;

handler leq<T> {
  solver EqualitySolver<T>;

  constraint leq(Logical<T>, Logical<T>) infix =<;

  rules {
    reflexivity  @ X =< X <=> true.
    antisymmetry @ X =< Y, Y =< X  # Id <=> X = Y pragma passive(Id).
    idempotence  @ X =< Y # Id \ X =< Y <=> true pragma passive(Id).
    transitivity @ X =< Y , Y =< Z ==> X =< Z.
  }
}
```

Listing 2: The `leq` handler with two, correct passive annotations.

```
X =< Z

| ?- Y =< Z, X =< Y.
Y =< Z,
X =< Y

| ?- Y =< Z, X =< Y, Z =< X.
Y = X,
Z = X
```

The last query shows that the handler would still be complete in the sense that all circular chains of leq-relations are collapsed into equalities. Nonetheless, adding the latter passive declaration is considered wrong: a pragma passive should only be used to help the compiler detect true passive occurrences – the code generated for these occurrences would be dead code anyway – and never to alter the behavior of a handler! In other words: it should be used as a optimization pragma, not as a control pragma.

FUTURE THOUGHTS

We say "help the compiler detect passive occurrences", and this is in fact still often the case in JCHR. The current version of the compiler does some symmetry analysis to detect passive occurrences, and will already detect several frequently occurring types, but when compared to the K.U.Leuven CHR System [SSD05a, SSD05b], the analysis is still quite naïve.

Asides from the notation illustrated in Listing 2 (the notation used by most other CHR systems), the K.U.Leuven JCHR System also offers following, more convenient variants for this popular pragma:

1. Multiple occurrences can be declared passive at once by using a comma-separated list of identifiers, e.g. `pragma passive(X, Y)`. The more cumbersome notation you have to use in other systems, `pragma passive(X), passive(Y)`, is also supported. **v1.3.1**

2. Occurrences can be declared passive directly in the head of a rule by adding '`# passive`'. This notation, which we refer to as *in-head*, is also adopted by the K.U.Leuven CHR System. **v1.4.0**

3. Because `passive` will probably remain the most popular pragma, it can be **v1.4.0**

written by simply adding a '`#`' after an occurrence (i.e. without an identifier or the `passive` keyword). This is indeed a very practical shorthand notation, which unfortunately, due to technicalities, was not possible to port to our Prolog CHR system.

Chapter 3

Types

Because the K.U.Leuven JCHR language is statically typed, types play an important role. A goal of the system has always been to make the integration of both languages, Java and CHR, as tight as possible. Previous CHR implementations in Java used either a form of typed logical variables [Sch99, AKSS01], or terms [Wol01], both syntactical entities well known from logical programming languages like Prolog. We considered this approach to be too limiting and counterintuitive for Java programmers (logical variables and terms should be possible, but the programmer should not be forced to use them; cf. also Section 3.2.4). We did not want to implement a Prolog-like CHR *in* Java, but rather a well embedded implementation of CHR *for* Java. Therefore, the variables you use in JCHR are *genuine Java variables*, and the typing rules resemble those known from Java.

To prevent the syntax from becoming too verbose, and to ease the porting from existing (non-Java) CHR systems, we did however introduce a form of *type coercion*. To be able to insert correct coercion code, the compiler needs some extra information. This knowledge is built-in for many common Java types (cf. Section 3.1), and the user can easily define and use its own variable types by annotating in with the necessary meta-data (more details follow in Section 3.2.

3.1 Built-in Types

The compiler implicitly knows all information it needs for most frequently used Java types:

1. The eight *primitive types*: `boolean`, `byte`, `short`, `int`, `char`, `long`, `float` and `double`.
2. The eight so-called wrapper types of the `java.lang` package: `Boolean`, `Byte`, `Short`, `Integer`, `Character`, `Long`, `Float` and `Double`.
3. `java.math.BigInteger` and `java.math.BigDecimal`
4. `java.lang.String`

Note that these types are all immutable values. We will see in Section 3.2.3 why this is significant.

Coercion

Coercion for the first two classes of built-in types boils down to the new auto-boxing/unboxing feature introduced in Java 5. It automates the cumbersome

switching between primitive types and their corresponding wrapper types. The result of all this magic is that you can largely ignore the distinction between e.g. `int` and `Integer`, with a few caveats. An `Integer` expression can have a `null` value. If `null` is coerced to `int`, a `NullPointerException` will be thrown. Secondly, there are some minor performance costs associated with boxing and unboxing.

Values of type `BigInteger` and `BigDecimal` are coerced to any primitive type, or any of the wrapper types.

3.2 User-defined Types

Asides from the built-in types listed in the previous section, the user can also define its own Java types and use them in the K.U.Leuven JCHR System. There are however some limitations. To be able to perform e.g. coercion, the compiler will need some extra information regarding these types. This is explained in Section 3.2.1. A more important problem (though it is a very convenient feature, coercion is mere syntactical sugar) is the so-called *modified problem*, described in detail in Section 3.2.2. In the section thereafter we sketch our proposed solution. We conclude with an example implementation of a user-defined type of logical variables (Section 3.2.4).

3.2.1 Type information

Coercion

Declaration/initialization

3.2.2 The modified problem

Important for an embedding of CHR in any host environment is the reactivation of suspended constraints, as soon as some guards (may) succeed. Crucial for efficiency, is that unnecessary reactivations occur as little as possible. However, we consider it important for an integration of CHR in an object oriented language like Java to allow so-called *behavioral matches* [BV94] in guards, besides the *structural matches* you encounter in most other CHR systems. Because objects are encapsulated entities, this leads to what in the literature is referred to as the *modified problem* [eoo94, Pac95, dFFR00]: how does the rule engine know when the state of an object has changed? Is this state change relevant? Which guards have to be re-evaluated? Things become even more involved if the inspector called in the guard depends on the state of multiple object. The latter is sometimes referred to as the *transitive modification problem* [dFFR00].

3.2.3 Observing modifications

The current solution we adopt is mainly based on the well-known observer pattern [GHJV95]. If you want to use a self-defined object type as an argument to JCHR constraints, this type has to be observable, i.e. it has to implement certain interfaces. This is explained in detail further in this section. There are two exceptions though: you do not have to implement the interfaces if your objects are immutable (cf. next paragraph), or if you use type modifiers, promising not to alter their states significantly (cf. Section 3.2.5).

Fixed types

If your objects are values that cannot be modified – i.e. there are no mutator methods that can change the state, at least not in a way that can ever influence the

result of some inspector methods – implementing these interfaces is not necessary. Indeed, the modified problem does not apply here as your objects can never be modified! We call such types *immutable types*, *value types*, or also *fixed types*. The Java types listed in Section 3.1, are all values: this is also implicitly know by the compiler. If you yourself have defined a reference type that you know is immutable, you can declare it to be so simply by annotating it with the annotation `annotations.JCHR_Fixed`.

Observable user-defined types

`runtime.Observable` As we said before, the modified problem is solved by requiring all (modifiable) types to implement the `runtime.Observable` interface. This is an application of the commonly-used observer design patter [GHJV95], where `runtime.Observable` plays the role of *observable*, `runtime.Constraint` of *observer* and the *notify method* is called `reactivate`. Constraints will register themselves as observers through the implemented method `addConstraintObserver`, and each time its state changes significantly, it is the responsibility of the observable to notify (reactivate) the observing constraints by calling their *reactivate* method.

Tip: If you want to use certain types, but you are not able to alter their source code, you can always try the decorator pattern [GHJV95] to create observable proxies for the objects involved that, after forwarding the mutator calls to the decorated object, reactivates the observing constraints.

Tip: You can use the `runtime.list.ConstraintLinkedList` to implement the list of observing constraints, certainly if you want the lists to be mergeable. An example usage can be found in `runtime.Logical`, explained further in Section 3.2.4.

`runtime.hash.HashObservable` Executing CHR is essentially performing many multi-way joins between constraints. For performance reasons the compiler will try to insert hash indices for partner constraint lookups. However, if your type is mutable, it is very well possible that also your hash value will change during the execution of a JCHR based program. To be able to keep the hash indices up-to-date, their keys will observe the objects contained in them, again using the observer design pattern. All mutable types have to implement the observer interface `runtime.hash.HashObservable`, through which keys will register themselves. It is the responsibility of an hash-observed object to notify all its observers using their `rehash` method each time its hash value has changed. An example implementation can be found in Section 3.2.4.

Tip: The tips given for `runtime.Observable` also apply here, except that this time `runtime.hash.MutableStorageKeySet` has been written to help with the implementation of the interface.

FUTURE THOUGHTS

The solution we adapt currently is still very ad-hoc:

- *It is too coarse-grained: it would be better to supply more information on the type of event that caused a constraint to be reactivated, and use this to deduce which guards have to re-evaluated. This would be something like the* wake conditions *introduced in [DSdlBH03].*
- *Currently all non-fixed types have to implement, besides the necessary* `Observable` *interface, also* `HashObservable`*. This is however only necessary for efficiency: all the compiler should do then is not to use hash-indices . . .*
- *Implementing the interface(s) requires a certain amount of work by the programmer. One idea we have to eliminate this cumbersome overhead would be the use of aspect orientation.*
- *We should allow java beans [Sun06a] with bound (and/or constrained) properties [Sun06b], certainly for those people who already have such bean classes in their existing applications.*

3.2.4 An example: `runtime.Logical`

3.2.5 Type modifiers

v1.1.0

There are several places in a JCHR handler where type declarations occur. The ones we are interested here are:

1. User-defined constraint declarations, explained in Section 2.6.1, contain type information for their arguments.
2. Variable declarations in the rules block (cf. Section 2.8.1).

From the preceding subsections we know that the JCHR runtime system needs to be informed about changes to the objects that are used in the arguments of the constraints it handles. We saw how the observer pattern [GHJV95] accomplishes this. But you might not want to implement the interfaces needed for this, if:

- you do not intend to modify the state of the objects, at least not in such a way that it is important for the runtime to know (i.e. it never affects the outcome of a guard)
- you do not want the rules to react to changes to object states for some reason
- the objects you are using are values, but you cannot add the `@\ac{JCHR}_Fixed` annotation (cf. previous section).

Therefore, you can add the *type modifier* '+' in front of the type to indicate that a certain argument of a user-defined constraint can be considered fixed by the compiler. Adding the '+' modifier to a type that is known to be a value – either because its declaration is annotated with `@\ac{JCHR}_Fixed`, as explained in Section 3.2.3, or because that knowledge is built into the compiler, as seen in Section 3.1 – is allowed, but of course not necessary.

FUTURE THOUGHTS

We could have the compiler detect that a certain argument is never used in a guard, because then it does not need to worry about the fact that its state might change. If this is the case, it could simply accept the typing without the need for a modifier.

Tip: Sometimes, if you know it will never change significantly, it is still interesting to add the modifier even to a type that is observable by the runtime: this can improve performance, because then observers will not be added and removed to the arguments.

FUTURE THOUGHTS

At the moment the only type modifier supported is '+'. If deemed interesting, future versions might incorporate other modifiers know from e.g. the K.U.Leuven CHR System, like '-' and '?'.

Be careful to use modifiers only if it is correct to do so: if the state of one of the constraint arguments does change, the runtime will have no knowledge about this event, and certain rules you might intended to fire, will not trigger.

Chapter 4

Built-in Constraints and Solvers

In Java CHR systems like JaCK [AKSS01, Sch99] and DJCHR [Wol01], there is only one real *built-in constraint* present (resp. equality of typed logical variables and equality of terms), very strongly coupled with the rest of the code. The same is in fact valid for pure Prolog: equality of terms can be seen as a built-in constraint, solved by the unification algorithm implemented by the Prolog host language. Inspired by the HAL CHR system [dlBDMS02, HdlBSD05], and the related paper [DSdlBH03], the K.U.Leuven JCHR System is designed to leave the choice of *built-in constraints* and their corresponding *built-in constraint solvers* as free as possible. It is relatively easy to add new built-in constraint solving capabilities, to switch to different implementations, to experiment with different variations, etc.

From Section 2.6.2 we already know how to declare the built-in solvers used by a particular handler. These declarations alone do not suffice, the compiler still needs some more information to decide which built-in solver to use for which built-in constraint, and which code to generate for each built-in constraint call. Our approach here is analogous to the one taken in Sections 3.1 and 3.2: the compiler knows the necessary information about all frequently used Java "constraints" and "solvers" (Section 4.2), and the user can define its own built-in constraint solvers using annotated meta-data (Section 4.3). Finally, Section 4.5 shows how JCHR handlers can be used as built-in solver to other JCHR handlers. But first, we need **v1.2.0** to tell something more about the two types of built-in constraints the K.U.Leuven JCHR System supports, namely *ask constraints* and *tell constraints*.

4.1 Ask versus Tell

In general, we distinguish three forms of interaction between a CHR system and built-in constraint solvers (see also [DSdlBH03]):

1. CHR adds new constraints to the built-in constraint solvers by firing rules: it *tells* the underlying solver certain constraints hold. These constraints are called *tell constraints*. All constraints in the body of a JCHR rule have to be tell constraints.

2. If we use constraints in a guard, we are typically *asking* the built-in solver whether or not certain constraints hold (i.e. logically entailed by the built-in constraint store). These constraints are called *ask constraints*. Although not part of any CHR specification [Frü98], some CHR systems also allow tell

Primitive types[a]		**Reference types**[b]		`Comparables`[c]	
Prefix	*Infix*	*Prefix*	*Infix*	*Prefix*	*Infix*
`eq`	`=` or `==`	`eq`	`=` or `==`		
`neq`	`!=`	`neq`	`!=`		
		`ref_eq`	`===`		
		`ref_neq`	`!==`		
`geq`	`>=`			`geq`	`>=`
`gt`	`>`			`gt`	`>`
`leq`	`<=` or `=<`			`leq`	`<=` or `=<`
`lt`	`<`			`lt`	`<`

Table 4.1: The built-in built-in ask constraints. They can all be written both prefix and infix.

[a] For `boolean` primitives, only equality and inequality can be asked.

[b] Object equality (`eq` and `neq`) is tested using the `Object.equals` method. For `enum` types, reference comparison (`==`) is always used. If you want to use reference comparison for other objects, you can use the `ref_eq` and `ref_neq` constraints. As in Java, reference (dis)equality can only be tested if the static types of the operands are comparable (possibly after coercion). You can for example not write `X === Y` if `X` is a `String` and `Y` is an `Integer`.

[c] "`Comparables`" are all types that (possibly after coercion) can be assigned to `java.lang.Comparable<T>`, numeric wrapper classes not included (these are first coerced to their corresponding primitive types). Commonly used `Comparable`s include `String`, `enum` types, `Boolean`, `BigInteger`, `Date`, etc.

constraints in guards. The K.U.Leuven JCHR System does not: all constraints in guards have to be ask constraints.

3. The final form of interaction is what we already encountered in Section 3.2.3: here it is the built-in constraint solver that takes the initiative, *notifying* the CHR system when some (and also as much as possible which) suspended CHR constraints should be reactivated. This is done when the built-in constraint store has changed in such a way that previously failed ask constraints might now be entailed. As we know from Section 3.2.3, this is more an interaction between the CHR system and the *variables* of the built-in constraint system.

It are the first two types of interaction we are interested in in the remainder of this chapter. We will see that a built-in constraint in JCHR can have either an *ask* version, a *tell* version, or both.

Example 4.1. Pure Prolog only has one built-in constraint solver, called the *Herbrand solver*, with as only tell constraint `=/2`, and corresponding ask constraint `==/2`. Even though many will use `=/2` in guards, behind the scenes, all Prolog CHR systems we know of (more or less) translate these guards to `==/2`.

4.2 Built-in built-in constraints

Table 4.1 shows all built-in built-in *ask* constraints known by the K.U.Leuven JCHR compiler. These constraints can only be asked, never told. The system also knows *assignment*, which, in a way, can be seen as a special tell constraint. This "constraint" can only be written infix, using the familiar '`=`' symbol, and its first argument has to be a variable. Its effect is only local (within a body): it is completely analogous to the assignment we know from Java. Although not enforced, an assignment is typically used to assign a value to a *local* variable.

Tip: For Java programmers: keep in mind that `==` (`!=`), does not test reference equality (disequality) of objects like you are used from Java. To test reference **v1.5.0**

```
package runtime;

import annotations.*;

@JCHR_Constraints({
    @JCHR_Constraint(
        identifier = "eq",
        arity = 2,
        ask_infix = {EQi, EQi2},
        tell_infix = EQi
    )
})
public interface EqualitySolver<T> {
    @JCHR_Tells("eq")
    public void tellEqual(Logical<T> X, T val);
    @JCHR_Tells("eq")
    public void tellEqual(T val, Logical<T> X);
    @JCHR_Tells("eq")
    public void tellEqual(Logical<T> X, Logical<T> Y);

    @JCHR_Asks("eq")
    public boolean askEqual(Logical<T> X, T val);
    @JCHR_Asks("eq")
    public boolean askEqual(T val, Logical<T> X);
    @JCHR_Asks("eq")
    public boolean askEqual(Logical<T> X, Logical<T> Y);
}
```

Listing 3: An annotated built-in solver interface (`EqualitySolver<T>`)

equality you have to append an extra `=` to the operators. This is done for reasons of symmetry with primitive types (cf. Table 4.1), and with Prolog built-in ask constraints (cf. Examples 4.1). It also turns out this is very close to a proposal to extend the Java language with similar syntactic sugar [Smi01].

4.3 User-defined built-in constraints

The approach chosen is similar to the one we took for user-defined types in Section 3.2: the user only has to declare which built-in constraint solvers are used in a certain handler (Section 2.6.2), and the compiler will use reflection on the annotated solver types to acquire all the necessary information. To be precise: the compiler needs to know which constraints the solver defines, and which methods it has to use to ask or tell these constraints.

Listing 3 shows an example of an annotated solver interface that defines one constraint. The annotations can also be put on a class declaration, rather than on an interface declaration. The only advantage of using an interface is, as always, implementation independence. A `@JCHR_Constraints` annotation is a comma-separated list of `@JCHR_Constraint`s, each declaring one, uniquely identified, constraint. A single solver class can easily define multiple constraints. Next to the `identifier` field, there is one other mandatory field, `arity`. For binary constraints, several other, optional fields are present, which we discuss in Section 4.3.1. Each method that should be used to tell (ask) one of the constraints is annotated with a

`@JCHR_Tells` (`@JCHR_Asks`) annotation, indicicating which of the solver's constraints it tells (asks).

Using this annotated meta-information together with the Java reflection facility, the K.U.Leuven JCHR compiler is capable of generating the correct method calls when encountering built-in constraints. You should never use the method calls directly. This is in line with the philosophy of declarative (constraint) programming. Not only does using the constraints improve ease of use and readability, it also makes it easier to switch to other solver implementations. Also, certain optimizations only work if problems are stated in terms of constraints (rather then host language statements). It would be possible to detect annotations on the methods that are used, but we have decided not to do so.

We already saw what to do if there are name clashes between constraints (between two built-in constraints, or between a built-in and a user-defined constraint) on page 18. Note that this will only occur if not only the constraint identifiers and arity are the same, but also their argument types. Most of the time, reflection on type information will also allow disambiguation.

4.3.1 Binary constraints

Binary constraints are in two ways special: firstly, they can, as is the case with binary user-defined constraints, be assigned one or more infix identifiers; secondly, the compiler can be provided with extra metadata on the properties of the constraint. **v1.5.0**

Declaration of `infix` identifiers **v1.5.0**

The declaration of `infix` identifiers is, analogous to what we saw in Section 2.6.1 for user-defined constraint declarations, optional, with analogous restrictions on the infix identifiers as well. The only exception is that the surrounding quotes are not allowed in the declaration[1]. If you use the identifier in a handler however, the accents are again mandatory (except for the ten built-in infix identifiers, cf. Section 2.6.1).

You can declare infix identifiers using different levels of granularity. You can:

- Declare one or more infix identifier for each constraint using the annotation's `infix` field, who's value is an array of `String`s. Singleton arrays can be written without the enclosing curly braces. This (these) infix identifier(s) can then be used both to tell and to ask the built-in constraint.
- Declare the infix identifiers used to aks and tell the constraint separately. This is illustrated in Listing 3. For this, the `JCHR_Constraint` annotation has two fields (both `String` arrays) `ask_infix` and `tell_infix`. You have to use either `infix`, or these two fields, or none at all. You can set a field to the empty array if you want (e.g. when infix identifiers should only be used to ask a particular constraint).
- You can even use different infix identifiers for each seperate method: the `@JCHR_Asks` and `@JCHR_Tells` annotations also include a field `infix`. The identifier(s) given here (or the empty array) override the default value given in the corresponding `@JCHR_Constraint` annotation. If you use this feature, the compact notation of the constraint identifier field – e.g. `@JCHR_Asks("eq")`,

[1] Also, if you would want to escape characters in the identifier, you will have to escape both the escaped character and the backslash. So, if you want to be able to use an infix identifier '\"my silly infix\"', you have to declare it as "\\\"my silly infix\\\""

which is actually short for `@JCHR_Asks(constraint="eq")`[2] – can no longer be used, so it has to be written in full: e.g. `@JCHR_Asks(constraint="eq", infix={"=", "=="})`.

Extra metadata **v1.5.0**

To help the K.U.Leuven JCHR compiler understand the properties of binary constraints, you should provide extra metadata. Below is a list of properties, and each of these properties corresponds to a field of the `@JCHR_Constraint` annotation. These fields can each have three values: `YES` (the constraint is known to have this property), `NO` (the constraint does not have this property), or `DEFAULT` (in which case the actual value is deduced from the prefix and infix identifiers of the constraint). Examples follow shortly, but first the definition of the properties. The items of this list are read as: *a binary constraint* `c` *is . . . , if and only if . . . holds.*

symmetric

$$\forall X, Y : \mathtt{c}(X, Y) \Rightarrow \mathtt{c}(Y, X)$$

asymmetric

$$\forall X, Y : \mathtt{c}(X, Y) \Rightarrow \neg\mathtt{c}(Y, X)$$

antisymmetric

$$\forall X, Y : \mathtt{c}(X, Y) \wedge \mathtt{c}(Y, X) \Rightarrow \mathtt{eq}(X, Y)$$

reflexive

$$\forall X, Y : \mathtt{eq}(X, Y) \Rightarrow \mathtt{c}(X, Y)$$

irreflexive

$$\forall X, Y : \mathtt{eq}(X, Y) \Rightarrow \neg\mathtt{c}(X, Y)$$

coreflexive

$$\forall X, Y : \mathtt{c}(X, Y) \Rightarrow \mathtt{eq}(X, Y)$$

total

$$\forall X, Y : \mathtt{c}(X, Y) \vee \mathtt{c}(Y, X)$$

transitive

$$\forall X, Y, Z : \mathtt{c}(X, Y) \wedge \mathtt{c}(Y, Z) \Rightarrow \mathtt{c}(X, Z)$$

trichotomous [3]

$$\forall X, Y : \mathtt{c}(X, Y) \oplus \mathtt{c}(X, Y) \oplus \mathtt{eq}(X, Y)$$

Notice the special status of the *equality* constraint `eq` in the above definitions. We say more about this special built-in constraint in Section 4.4. The following example should help make clear the difference between equality constraints, and coreflexive constraints:

Example 4.2. Reference comparison of objects in Java should, in general, not be considered an equality constraint. It is on the other hand often safe to regard comparing objects using the `Object.equals` method as an equality constraint (cf. Section 4.4). It is not because two objects are distinct, in the sense that they have different object identities, they are not "equal". The following snippet of Java code should clarify this:

[2] Actually, due to restrictions of the Java language, it is short for `@JCHR_Asks(value="eq")`, but this hidden field is in fact not intended to be used in full. We cannot prohibit it, but we advice to always use the `constraint` field.

[3] We let $\oplus$ denote *exclusive disjunction*, i.e. $(P \oplus Q) \Leftrightarrow ((P \vee Q) \wedge \neg(P \wedge Q))$

```
Integer oneFive = new Integer(5);
Integer otherFive = new Integer(5);

if (oneFive != otherFive)
    System.out.println("They do not reference the same object, ...";
if (oneFive.equals(otherFive))
    System.out.println("... but they are equal nontheless!");
```

Reference identity does however imply equality, or, in other words, reference comparison is a coreflexive constraint. Or to be more precise: it is coreflexive, as long as the implementor of e.g. the *equals* method obeys its general contract specified in the Java API. But this is an assumption we make, as is specified in Section 4.4.

It is, as shown in Listing 3, for most common types of constraints not needed to specify all these properties, as long as you obey the implicit naming conventions for their identifiers. The default value of the fields is – prepare yourself for a big surprise – `DEFAULT`. This means the actual value will be derived from the prefix or infix identifier of the constraint. We will clarify this with one example, but similar, equally intuitive rules apply for all the above properties, and for all the following constraint identifiers: prefix: `eq`, `neq`, `leq`, `lt`, `geq`, `gt`; and infix: `=`, `==`, `===`, `!=`, `!==`, `<=`, `=<`, `>=`, `<`, `>` (note that these are the ten built-in infix identifiers seen on page 15). All concrete rules are also specified in the `API` of `compiler.CHRIntermediateForm.constraints.bi.BuiltInConstraint`.

Example 4.3. To determine whether a constraint is symmetric by `DEFAULT`, the following rules are applied (in the enumerated order):

1. If the prefix identifier of the constraint is either of the following values, the default value is `true`: `eq`, `leq`, `geq`.
2. If the infix identifier of the constraint is either of the following values, the default value is `true`: `=`, `==`, `<=`, `=<`, `>=` for all other constraints the default value is `false`.

4.3.2 `java.util.Comparator` solvers

Subtypes implementing the `java.util.Comparator<T>` interface are a special case. These types can always be used as a built-in constraint solver, even if no *explicit* annotations are present, because they already *implicitly* define four built-in constraints. These four constraints are the same as the four in the "Comparables" subtable of Table 4.1. Even though they are not (and cannot) be declared using annotations, they are treated (e.g. in case of name clashes) as (user-defined) built-in constraints. It is always possible to declare extra constraints solved by `Comparator` constraint solvers, as long as the constraint identifiers are different than the ones of the four implicit constraints.

4.4 Equality constraints

Equality constraints are special for several reasons:

1. We already saw in Section 4.3.1 that they occupy a special position when reasoning about the properties of (binary) constraints. **v1.5.0**
2. The compiler has to be able to use them to generate implicit guards, as seen in Section 2.8.4. Therefore, we have fixed its identifier: if you want your constraint to be used to resolve implicit equality guards, you have to give it the

prefix identifier `eq` (and of course provide an ask version). Not that we intend to change this, but it might be safest to use the `IBuiltInConstraint.EQ` constant for this. Although it is common also to use `=` or `==` as infix identifier, this is *not* required: when it encounters an implicit guard, the compiler will only look for a suited ask built-in constraint with prefix identifier `eq`.

3. If the compiler uses hash indices, it relies on certain properties (transitivity and symmetry to be precise) of equality constraints. We will give properties each equality constraint should have below.

Required properties

Each equality constraint should be:

- reflexive (cf. Section 4.3.1 for a definition)
- symmetric (cf. Section 4.3.1 for a definition)
- transitive (cf. Section 4.3.1 for a definition)
- consistent: asking the constraint more than once (e.g. multiple invocations of the `equals` method), consistently return the same result, provided no information used in the comparison has changed. If the object is modified in such a way that it (can) affect the entailment of an equality constraint, the built-in constraint solver has to notify the necessary JCHR handlers (cf. Sections 3.2.3 and 4.1).
- hash-consistent: if two objects are equal, they also have the same hash value (as returned by the `hashCode` method.

The *equals* method

We saw in Section 4.2 that the K.U.Leuven JCHR compiler treats the `Object.equals` method as a built-in ask equality constraint. This is in fact only safe as long as both arguments are fixed (cf. Section 3.2.3), and the implementor of the method obeys the general contract specified in the Java API of this method. If this is done, all properties listed in the above subsection indeed hold.

4.5 JCHR handlers as built-in solvers

v1.2.0

You can use a (compiled) JCHR handler as a built-in solver to another JCHR handler. For this, generated handler classes (cf. Section 5.2.2) contain the necessary annotations to declare them built-in constraint solver of some other handler (cf. Section 2.6.2). User-defined JCHR constraints can only be told, never asked. The JCHR constraints of a handler declared as built-in solver can also never be used in the head of a rule.

FUTURE THOUGHTS

One might wonder why JCHR constraints cannot be asked. The reason is that, in general, this is a complicated problem. The fact that the asked constraint is not present in a CHR constraint store does not necessarily mean the constraint is not logically entailed *by the combined state of the CHR system and and the built-in stores. The problem is addressed in [SDD+05b, SDD+05a].*

Chapter 5

Compiling a K.U.Leuven JCHR Handler

The K.U.Leuven JCHR compiler compiles a K.U.Leuven JCHR handler to a series of Java source files. These are then typically[1] compiled to Java byte code by a third party Java compiler. Together with the classes present in the K.U.Leuven JCHR runtime system, the compiled handler code then forms an efficient and stable constraint system.

5.1 Requirements

The K.U.Leuven JCHR System is written using the Java 2 Standard Edition (J2SE) 5.0 platform [Sunb]. So to compile a K.U.Leuven JCHR handler you need to be able to run programs written in that version of the Java language. You will also need a Java SDK containing a compiler capable of compiling the generated Java files (the compiler will also have to be Java 5.0 compatible).

By far the most widely used J2SE platform is Sun Microsystems' reference implementation, referred to as the J2SE Runtime Environment (JRE). Sun also offers a Java compiler, contained in the JDK, their reference implementation of a Java SDK[2]. Both software packages are available for free at Sun's Java website [Sunb]. Note that the JDK installation program will also ask whether you want to install the most recent JRE.

Asides from a suited Java platform, you also need the following Java libraries[3]:

- ANTLR Parser Generator v2 (version 2.7.5 or higher) [P+06]
- FreeMarker Template Engine (version 2.3 or higher) [fre06]
- Args4j (version 2.0.4 or higher!) [Kaw06]

All these tools offer pre-compiled jar files, that can easily be added to your class search path.

[1] There also exists compilers that compile e.g. to machine code...

[2] If you are using the Sun Java compiler, we advice to use J2SE Development Kit (JDK) 1.5.0 update 6 or higher. Earlier version have a bug that causes some unnecessary warnings to be generated [Sun05].

[3] Though never tested, earlier versions of these libraries might also work. Any higher, backwards compatible version will work.

5.2 Compilation

First make sure you have a suited Java platform, as described in the previous section, and that the third party libraries listed there are included in the class search path. We assume in this section that you are using Sun's JDK, and that all libraries are already included (e.g. using the CLASSPATH environment variable). For more information on how to run a Java program, we refer to appendix B.

To compile a handler file called `xxx.jchr`, a typical session starts with:

```
java compiler.Main < xxx.jchr
```

(providing a single input file through the standard input stream), or: **v1.4.0**

```
java compiler.Main xxx.jchr
```

Using the latter method, you can also compile multiple JCHR files at once: **v1.4.0**

```
java compiler.Main xxx.jchr yyy.jchr
```

Combining these two input methods is not possible. If you provide one or more file names, the default input stream will be ignored. Only if no file names are provided, the compiler will check the standard input stream. If this stream is also empty, the compiler assumes no source file was provided, and print a usage overview. There exists a compiler option telling the compiler to block, waiting for input to arive over the standard input stream. More information on compiler options is provided in the next subsection.

Now, say the handler in `xxx.jchr` is declared part of package `zzz.yyy.xxx` (cf. Section 2.3), then the compiler will generate several Java source files in the directory `./zzz/yyy/xxx`. More details on generated files follow in Section 5.2.2. These files **v1.5.0** then have to be compiled by your preferred Java compiler. This is done using an instruction that looks like this:

```
javac zzz/yyy/xxx/*.java
```

It is possible (cf. Section 5.2.2) some extra generated helper classes will also have to be compiled:

```
javac runtime/Tuple*.java
```

Note that for compiling the generated files only the K.U.Leuven JCHR runtime system is required on the class path, i.e. no third party libraries (or the K.U.Leuven JCHR compiler). The same applies when using the resulting classes in your applications afterwards (cf. Section 6).

5.2.1 Compiler options

The command line compiler tool also accepts compiler options:

Name	Values	Default	Description
`hash`	boolean	`true`	Toggles use of hash indices
`debug`	`off/default/full`	`default`	See Section 7.1
`standardinput`	/	/	Toggles blocking input

Options are used as follows:

```
java compiler.Main {-optionname optionvalue}* ...
```

Valid values for boolean options are: `on/off`, `true/false` and `yes/no`. The `standardinput` option is an example of a flag option: it does not take a value, it is just either present or not. If the latter flag is present, the compiler will block waiting for input to arive over the standard input stream, as explained above.

5.2.2 Generated code

v1.5.0

For a source file declaring a handler h (this is the name given in the handler declaration, which is not necessarily the same as the one used in the filename) the compiler generates one Java source file called h`Handler.java`. This file will be generated in the correct directory: if your handler is declared to be part of package `org.foo.bar` (cf. Section 2.3), the file will be generated in the corresponding directory `./org/foo/bar/`. If this directory does not exist, the compiler will try to create it (this should not happen if you follow the proposed convention of keeping the handler source file in this same directory).

The compiler might generate some extra utility class files `./runtime/tuples/Tuple`n`.java` with n a natural number. These file are part of the generic runtime code, and can be used by multiple JCHR handlers.

If the handler declares user-defined constraints c_1, ..., c_n, this source file will contain the code for $n + 1$ classes:

- One top-level class, h`Handler`. The access modifier of this handler is the same as the one preceding the handler declaration in the handler source file (cf. Section 2.5).
- For each user-defined constraint c_i an inner class c_i`Constraint.java`. The access modifier of a constraint class depends on the access modifier of the corresponding constraint declaration (cf. Section 2.6.1).

More information on the generated classes and how to use them is given in the next chapter.

Chapter 6

Using a K.U.Leuven JCHR Handler

6.1 Requirements

To use a K.U.Leuven JCHR handler you only need the compiled files generated by the K.U.Leuven JCHR compiler (cf. Section 5), and the K.U.Leuven JCHR runtime system. The requirements for the Java platform are the same as described in Section 5.1, except that you do not need a Java compiler (so e.g. the JRE would suffice). Also, no third-party tool libraries are needed to use a JCHR handler.

6.2 Stand-alone

A compiled K.U.Leuven JCHR handler cannot be executed as a stand-alone program, but is intended to be used integrated in another Java program.

FUTURE THOUGHTS

- *We might someday add goal sections, like the ones used in JaCK [AKSS01, Sch99]. These are short snippets of code (similar to the body of a rule), allowing you to encode simple queries, and would mainly be intended for experimentation and debugging.*
- *An orthogonal alternative would be to simply allow a main method declaration, allowing you to write any query, but not in a declarative way as in a goal section.*
- *Thirdly, it might be interesting (not only to become stand-alone, but also to initialize a handler), to allow propagation rules without a head. This would be equivalent with a single goal section, with that difference that it is* also *executed if used embedded in other code, whereas goal sections are only executed if used in stand-alone mode.*

6.3 Integration with Java

The K.U.Leuven JCHR System allows for an easy and direct integration of the constraint system into a Java application or applet. As a running example, we will be using the simple Java program listing on page 42.

The first thing to do is *compile* the handler(s) you want to use (as explained in Section 5) and *import* the generated Java classes (cf. Section 5.2.2). If, as in our example, the `Handler` class is part of the same package (cf. line 1) as the current

```
package examples.gcd;

import java.util.Collection;

import examples.gcd.GcdHandler.GcdConstraint;

public class Gcd {

public static void main(String[] args) throws Exception {
   if (args.length != 2) printUsage();
   else try {
      final long i0 = Long.parseLong(args[0]),
                 i1 = Long.parseLong(args[1]);

      if (i0 < 0 || i1 < 0) {
         printUsage();
         return;
      }

      // First we create a new JCHR constraint handler:
      GcdHandler handler = new GcdHandler();

      // Next we tell the JCHR handler the following two constraints:
      handler.tellGcd(i0);
      handler.tellGcd(i1);

      // Afterwards we can lookup the constraints in the
      // resulting constraint store:
      Collection<GcdConstraint> gcds = handler.getGcdConstraints();
      long gcd;

      // There should be exactly one constraint, containing
      // the greatest common divider:
      assert gcds.size() == 1;

      gcd = gcds.toArray(new GcdConstraint[1])[1].get$0();

      // Simply print out the result:
      System.out.printf(" ==>  gcd(%d, %d) == %d", i0, i1, gcd);

   } catch (NumberFormatException e) {
      System.err.println(e.getMessage());
      printUsage();
   }
}

public final static void printUsage() {
   System.out.println(
      "Usage: java Gcd <positive int> <positive int>"
   );
}
}
```

```
1  PrimesHandler handler = new PrimesHandler();
2  handler.tellCandidate(1000);
3  Filter<PrimeConstraint> fltr = new InclusionFilter<PrimeConstraint>(
4    @Override
5    public boolean hasToInclude(PrimeConstraint constraint) {
6      return (constraint.get$0() > 123)
7          && (constraint.get$0() < 321);
8    }
9  );
10 for (PrimeConstraint constraint : handler.getPrimeConstraints(fltr))
11   System.out.println(constraint);
```

Listing 4: Queriing the constraint store of a `PrimesHandler` for prime numbers between 123 and 321 using a user-defined filter.

compilation unit, no import is of course necessary. The only constraint class for this handler is imported on line 5. Since `Constraint` classes are always inner classes, they do have to be imported explicitly. **v1.5.0**

Next, you typically create the *handler* objects you need. In our example this is done on line 21. Not illustrated is that a JCHR handler's constructor takes all its declared built-in solvers as an argument (in the order they were declared, cf. Section 2.6.2).

Thirdly, one *tells* a series of constraints to the solver. Of course, if built-in solvers are present, also built-in constraints can be told or asked in between.

At any time, the current constraint JCHR store can be inspected. This is because CHR is an *incremental* language. For each user-defined constraint C, a handler class offers:

- a `lookup`C`()` method, returning a `java.util.Iterator` over all C constraints.
- a `get`C`Constraints()` method, which returns a `java.util.Collection` containing all C constraints. **v1.3.1**
- a `get`C`Constraints(Filter)` method, which returns a filtered view of the C constraints in the current store, using a user-defined filter (i.e. an instance of `util.iterator.Filtered.Filter`). Note that you cannot directly subclass the `Filtered.Filter` class, instead you should extend one of its subclasses `InclusionFilter`, `ExclusionFilter`, `IndexInclusionFilter` or `IndexExclusionFilter`. **v1.5.0**

The access modifier of these methods depends on the access modifier given in the corresponding constraint declaration (cf. Section 2.6.1). In fact, for *private* constraints, these methods might not even be generated. **v1.5.0**

Example 6.1. If you want to iterate over all prime numbers between 123 and 321, you can use the Java code from Listing 4.

Also, each handler object itself is a true collection of `runtime.Constraints`, in the sence that it implements the Java interface `java.util.Collection<Constraint>`. **v1.3.2**
So, you can e.g. use the handler in a for-each loop somewhere in your Java code:

```
for (Constraint constraint : handler) {
   ...
}
```

An equivalent to the `iterator` method is the (legacy) `lookup` method. Both return **v1.3.1**
an iterator over all *accessible* constraints in the constraint store. For the definition of 'accessible', see below. It is also possible get a filtered view (cf. supra) of all *accessible* constraints at once using the `Handler.filter(Filter)` method. **v1.5.0**

To preserve encapsulation, non-`public` (cf. Section 2.6.1) constraints are not al- **v1.5.1**
ways included in the `Collection` the handler implements. In other words: they are not automatically included in the results of methods like `iterator()` and `size()`. That is what we mean by 'accessible' in the above paragraph. We distinguish two cases:

- For *public* handlers, only *public* and `local` constraints are accessible. For those classes who have access to other constraints as well, the generated code of *public* handlers will contain two extra methods:

  `includeProtected()` This `protected` method returns a handler object of the same static type as the instance it is called on. The result of this method will then include all `protected` constraints in its collection.

  `includePackage()` Same as `includeProtected()`, but includes constraints with default access as well.

  Example 6.2. To iterate over all non-`private` constraints of a handler, you can use:

  ```
  for (Constraint constraint : handler.includePackage())
          System.out.println(constraint);
  ```

  Of course, this method has default access, and can thus only be called from within the same package.

- For handlers with default accessibility, only *private* constraints are excluded.

Note that in this scheme `private` constraints are indeed never exposed.

Tip: When using the more trusted versions of handlers (i.e. non-`public` handlers, or the handlers returned by the `include*` methods), care should be taken not to pass these instances to less trusted code. If such handler instances are leaked, constraints with restricted access become visible for untrusted code. This should not occur though, as these handlers can only be used by trusted code in the first place (and code that leaks restricted information is not trustworthy)!

Chapter 7

Debugging a K.U.Leuven JCHR Handler

Because handlers are compiled to Java code, you can simply use any tool you normally use to debug Java. To help you understand the runtime behavior of a handler, the K.U.Leuven JCHR System also offers a trace debugger.

7.1 Trace Debugger

v1.4.0

K.U.Leuven JCHR offers a simple tool for reasoning explanation: it is possible to register listeners for important JCHR events. The current implementation is in a limited, experimental stage, but is already very helpful for debugging and understanding JCHR execution. Currently, events can be thrown on the following execution points:

- Right after a constraint is activated (for the first time).
- Right after a constraint is reactivated (by some built-in event).
- Right after a constraint is stored in the constraint store.
- Right after a constraint is removed from the constraint store.
- Right after a constraint is terminated. **v1.5.0**
- Right *before* a rule fires. Arguments passed along with the event also include the constraints that matched the head and the index of the active constraint.

The event listener interface is called `runtime.debug.Tracer`. Each handler can have one `Tracer`, which has to be provided at its construction, or later through the `setTracer` method. **v1.5.0**

Tip: `runtime.debug.SysoutTracer` provides an implementation of the `Tracer` interface that prints all events to the default output stream. This is probably the tracer that will be used most of the time for now. A similar tracer, called `runtime.debug.FileTracer`, logs all events to a user-defined file. **v1.5.0**

Tip: Another simple tracer, `runtime.debug.StatisticsTracer`, collects some basic runtime statistics like number of constraint activations, rule firings, etc. These could be helpful to find e.g. performance bottlenecks. **v1.5.0**

Tip: If you want the events to be distributed to different listeners, you can use the `runtime.debug.CompositeTracer` class.

FUTURE THOUGHTS

We are working on a visual tracer, showing a view of the constraint store. It already works, but there is still some work to be done. Future versions of this visual tracer might incorporate views of the constraint activision stack, since the constraint store does not provide all required runtime information to effectively debug a JCHR program.

Code generation for throwing these events is controlled with the *debug* option and the *debug* pragma. The `debug` option can be set command-line or in the JCHR source file, and has three valid values:

`off` No code for (debug) tracing will be generated.

`default` This is the default value: code to add a tracer to a handler will be generated. The only event told are rule-firing, and only for rules annotated with `pragma debug`. If no rule is annotated, no tracer code at all is generated. I.e. by default none of the generated code is affected by this new feature. This is done to avoid performance penalties.

`full` More events are raised then in the default case: all constraint activations, reactivations, additions to and removals from the constraint store also result in events. Also, all rule firings will raise events, independent of possible pragmas.

FUTURE THOUGHTS

The way trace settings are set is likely to change in future versions, as well as the events that are raised. It would be e.g. interesting to integrate with the built-in solvers, to have a more fine-grained control on which events should be raised, etc.

Appendix A

Example programs

This appendix contains several example programs referred to throughout the text. More examples can be found on the K.U.Leuven JCHR website [VW06].

(1) primes This is not really a typical constraint solver, but a first example of the use of CHR as "general purpose" Constraint Programming (CP)-language. It also illustrates how primitive Java values are used naturally in JCHR.

(2) leq The leq-handler is the prototypical example of a generic handler. It also shows how infix notation can improve readability. It is well-known in the CHR literature that this handler all nice properties like confluence, completeness, etc.

(3) fib This program clearly shows one of the advantages of using Java as a host language: the availability of a rich type library. The `BigInteger` class allows you to represent arbitrary large integer numbers in a simple and efficient way. The handler also illustrates how primitive types and logical variables can be used side by side quite elegantly.

(4) gcd This example is similar to 1. It shows how one can very easily implement an algorithm – in this case a variant of Euclid's algorithm for computing greatest common dividers – using CHR.

(5) guf A second, larger example of the power of generic handlers is this generic version of the union-find handler: the transition of untyped Prolog to strongly typed Java is eased significantly. This handler also illustrates once more that it can be very interesting to use logical variables[1], Java-variables and even primitive variables side by side.

(6) iuf This is the same solver as **guf**, but then for primitive integer-variables (Java generics does not extend to primitive types). The acronym iuf stands for integer union-find. Note that without generics, one would have to make such an adjustment for each type, not only for primitive types!

(7) bool Even though this pure constraint solver is somewhat larger than the other examples, it nevertheless is just as intuitive, flexible and efficient!

[1] Do not forget that these are just Java-variables of some specific type, implementing logical variables!

Example program A.1: the primes-handler

```
package examples;

import static util.arithmetics.primitives.intUtil.*;

public handler primes {
        public constraint candidate(int);
        local constraint prime(int);

        rules {
                candidate(1) <=> true.
                candidate(N) <=> prime(N), candidate(dec(N)).

                absorb @ prime(Y) \ prime(X) <=> modZero(X, Y) | true.
        }
}
```

Example program A.2: the leq-handler

```
package examples.leq;

import runtime.*;

public handler leq<T> {
        solver EqualitySolver<T>;

        public constraint leq(Logical<T>, Logical<T>) infix =<;

        rules {
                reflexivity  @ X =< X <=> true.
                antisymmetry @ X =< Y, Y =< X <=> X = Y.
                idempotence  @ X =< Y \ X =< Y <=> true.
                transitivity @ X =< Y, Y =< Z ==> X =< Z.
        }
}
```

Example program A.3: the fib-handler

```
package examples.fib;

import java.math.BigInteger;
import runtime.Logical;
import static util.arithmetics.primitives.intUtil.*;

public handler fib {
    solver runtime.EqualitySolver<BigInteger>;

    public constraint fib(int N, Logical<BigInteger> M);

    rules {
        fib(N,M1), fib(N,M2) <=> M1 = M2, fib(N, M1);

        fib(0,M) ==> M = 1;

        fib(1,M) ==> M = 1;

                local int N1, N2;
                local Logical<BigInteger> M1, M2;

        fib(N,M) ==> N > 1 |
            N1 = dec(N),    fib(N1,M1),
            N2 = sub(N, 2), fib(N2,M2),
            M = M1.add(M2);
    }
}
```

Example program A.4: the gcd-handler

```
package examples.gcd;

import util.arithmetics.primitives.longUtil;

public handler gcd {
        public constraint gcd(long);

        rules {
                gcd(0) <=> true.
                gcd(N) \ gcd(M) <=> M >= N | gcd(longUtil.sub(M, N)).
        }
}
```

Example program A.5: the guf-handler

```
package examples.unionfind;

import runtime.Logical;
import static util.arithmetics.primitives.intUtil.*;

public handler guf<E> {
        solver runtime.EqualitySolver<E>;

        public constraint make(+E),
                union(+E, +E), find(+E, Logical<E> Result);

        private constraint root(+E Root, int Rank),
                link(+E Root1, +E Root2),
                arrow(+E Node, +E Parent) infix '~>';

        rules {
                local Logical<E> R_x, R_y;

                make @ make(X) <=> root(X,0);

                union @ union(X, Y) <=> find(X, R_x), find(Y, R_y), link(R_x, R_y);

                // path compression with immediate update thanks to logical variable
                findNode @ X '~>' Parent_x, find(X, R) <=> find(Parent_x, R), X '~>' R;
                // return function result in second argument
                findRoot @ root(X, _) \ find(X, R) <=> R = X;     /* found */

                // root treatment
                linkEq   @ link(R, R) <=> true.
                linkLeft @ link(R_x, R_y), root(R_x, Rank_x), root(R_y, Rank_y) <=>
                        Rank_x >= Rank_y | R_y '~>' R_x, root(R_x, max(Rank_x, inc(Rank_y)));
                linkRight@ link(R_y, R_x), root(R_x, Rank_x), root(R_y, Rank_y) <=>
                        Rank_x >= Rank_y | R_y '~>' R_x, root(R_x, max(Rank_x, inc(Rank_y)));
        }
}
```

Example program A.6: the iuf-handler

```
package examples.unionfind;

import runtime.primitive.LogicalInt;
import static util.arithmetics.primitives.intUtil.*;

public handler iuf {
        solver runtime.primitive.IntEqualitySolver;

        public constraint make(int),
                union(int, int), find(int, LogicalInt Result);

        private constraint root(int Root, int Rank),
                link(int Root1, int Root2),
                arrow(int Node, int Parent) infix '~>';

        rules {
                local LogicalInt R_x, R_y;

                make @ make(X) <=> root(X,0);

                union @ union(X, Y) <=> find(X, R_x), find(Y, R_y), link(R_x, R_y);

                // path compression with immediate update thanks to logical variable
                findNode @ X '~>' Parent_x, find(X, R) <=> find(Parent_x, R), X '~>' R;
                // return function result in second argument
                findRoot @ root(X, _) \ find(X, R) <=> R = X;     /* found */

                // root treatment
                linkEq   @ link(R, R) <=> true.
                linkLeft @ link(R_x, R_y), root(R_x, Rank_x), root(R_y, Rank_y) <=>
                        Rank_x >= Rank_y | R_y '~>' R_x, root(R_x, max(Rank_x, inc(Rank_y)));
                linkRight@ link(R_y, R_x), root(R_x, Rank_x), root(R_y, Rank_y) <=>
                        Rank_x >= Rank_y | R_y '~>' R_x, root(R_x, max(Rank_x, inc(Rank_y)));
    }
}
```

Example program A.7: the bool-handler

```
package examples;

import runtime.primitive.LogicalBoolean;

public handler bool {
        solver runtime.primitive.BooleanEqualitySolver;

        public constraint
                and(LogicalBoolean X, LogicalBoolean Y, LogicalBoolean Result),
                or(LogicalBoolean X, LogicalBoolean Y, LogicalBoolean Result),
                xor(LogicalBoolean X, LogicalBoolean Y, LogicalBoolean Result),
                neg(LogicalBoolean X, LogicalBoolean Result),
                imp(LogicalBoolean X, LogicalBoolean Y);

        option(hash, false);

        rules {
                /*
                 * and/3 specification
                 *        and(0,0,0).
                 *        and(0,1,0).
                 *         and(1,0,0).
                 *         and(1,1,1).
                 */

                and(false,X,Y) <=> Y=false;
                and(X,false,Y) <=> Y=false;
                and(true,X,Y) <=> Y=X;
                and(X,true,Y) <=> Y=X;
                and(X,Y,true) <=> X=true,Y=true;
                and(X,X,Z) <=> X=Z;
                //and(X,Y,X) <=> imp(X,Y);
                //and(X,Y,Y) <=> imp(Y,X);
                and(X,Y,A) \ and(X,Y,B) <=> A=B;
                and(X,Y,A) \ and(Y,X,B) <=> A=B;

                /*
                 * or/3 specification
                 *         or(0,0,0).
                 *         or(0,1,1).
                 *         or(1,0,1).
                 *         or(1,1,1).
                 */

                or(false,X,Y) <=> Y=X;
                or(X,false,Y) <=> Y=X;
                or(X,Y,false) <=> X=false,Y=false;
                or(true,X,Y) <=> Y=true;
                or(X,true,Y) <=> Y=true;
                or(X,X,Z) <=> X=Z;
                //or(X,Y,X) <=> imp(Y,X);
```

```
//or(X,Y,Y) <=> imp(X,Y);
or(X,Y,A) \ or(X,Y,B) <=> A=B;
or(X,Y,A) \ or(Y,X,B) <=> A=B;

/*
 * xor/3 specification
 *   xor(0,0,0).
 *   xor(0,1,1).
 *   xor(1,0,1).
 *   xor(1,1,0).
 */

xor(false,X,Y) <=> X=Y;
xor(X,false,Y) <=> X=Y;
xor(X,Y,false) <=> X=Y;
xor(true,X,Y) <=> neg(X,Y);
xor(X,true,Y) <=> neg(X,Y);
xor(X,Y,true) <=> neg(X,Y);
xor(X,X,Y) <=> Y=false;
xor(X,Y,X) <=> Y=false;
xor(Y,X,X) <=> Y=false;
xor(X,Y,A) \ xor(X,Y,B) <=> A=B;
xor(X,Y,A) \ xor(Y,X,B) <=> A=B;

/*
 * neg/2 specification
 *   neg(0,1).
 *   neg(1,0).
 */

neg(false,X) <=> X=true;
neg(X,false) <=> X=true;
neg(true,X) <=> X=false;
neg(X,true) <=> X=false;
neg(X,X) <=> fail;
neg(X,Y) \ neg(Y,Z) <=> X=Z;
neg(X,Y) \ neg(Z,Y) <=> X=Z;
neg(Y,X) \ neg(Y,Z) <=> X=Z;
// Interaction with other boolean constraints
neg(X,Y) \ and(X,Y,Z) <=> Z=false;
neg(Y,X) \ and(X,Y,Z) <=> Z=false;
neg(X,Z) , and(X,Y,Z) <=> X=true,Y=false,Z=false;
neg(Z,X) , and(X,Y,Z) <=> X=true,Y=false,Z=false;
neg(Y,Z) , and(X,Y,Z) <=> X=false,Y=true,Z=false;
neg(Z,Y) , and(X,Y,Z) <=> X=false,Y=true,Z=false;
neg(X,Y) \ or(X,Y,Z) <=> Z=true;
neg(Y,X) \ or(X,Y,Z) <=> Z=true;
neg(X,Z) , or(X,Y,Z) <=> X=false,Y=true,Z=true;
neg(Z,X) , or(X,Y,Z) <=> X=false,Y=true,Z=true;
neg(Y,Z) , or(X,Y,Z) <=> X=true,Y=false,Z=true;
neg(Z,Y) , or(X,Y,Z) <=> X=true,Y=false,Z=true;
neg(X,Y) \ xor(X,Y,Z) <=> Z=true;
neg(Y,X) \ xor(X,Y,Z) <=> Z=true;
neg(X,Z) \ xor(X,Y,Z) <=> Y=true;
```

```
                neg(Z,X) \ xor(X,Y,Z) <=> Y=true;
                neg(Y,Z) \ xor(X,Y,Z) <=> X=true;
                neg(Z,Y) \ xor(X,Y,Z) <=> X=true;
                neg(X,Y) , imp(X,Y) <=> X=false,Y=true;
                neg(Y,X) , imp(X,Y) <=> X=false,Y=true;

                /*
                 * imp/2 specification (implication)
                 *   imp(0,0).
                 *   imp(0,1).
                 *   imp(1,1).
                 */

                imp(false,X) <=> true;
                imp(X,false) <=> X=false;
                imp(true,X) <=> X=true;
                imp(X,true) <=> true;
                imp(X,X) <=> true;
                imp(X,Y),imp(Y,X) <=> X=Y;

        }
}
```

Appendix B

Running a Java Program

This section contains some basic pointers on how to run Java programs. Of course is focused on what is needed to use the K.U.Leuven JCHR System, for more detailed information you should consult the manual of the Java platform you are using. We refer to sections 5.1 and 6.1 for the required version of the Java platform. The main focus here will be the use of Sun's distribution of Java (JRE and JDK) [Sunb]. If you are using another Java platform implementation, or some Java IDE (like Eclipse), then you should look at their respective documentations for information.

B.1 Verifying Java Platform version

To verify which version of the Java platform (if any) is installed on your system, you typically type on the command line:

```
java -version
```

Because you will probably also need to compile the generated Java files (cf. Section 5), you do not only need a suited JRE but also the JDK, containing the basic tools required to create Java applications, like a Java compiler. You can check whether a suited version of the compiler is installed like this:

```
javac -version
```

The Sun reference distribution can be downloaded freely at [Sunb]. Note that when installing the JDK, the installation program will also whether you want to install the JRE as well.

B.2 Setting the Java class search path

When the Java virtual machine runs your program, it searches for application `.class` files using the paths listed in the *class search path*. The compiler searches for required components the same way. By default they look for classes in the current working directory. However, if your program uses classes that are not in the current working directory, you need to explicitly list all the directories containing classes used by the application. In addition, if your program uses classes contained in a `jar` file, then that file must be listed in the class search path as well.
There are (at least) two ways to specify the class search path:

1. Using the `classpath`-option of the program, e.g.

   ```
   java -classpath ***** SomeClassContainingMain
   ```

and like ways:

```
javac -classpath ***** SomeSourceFile.java
```

2. Using a `CLASSPATH` environment variable. For a Windows system:

   ```
   set CLASSPATH=*****
   ```

   For a Linux system:

   ```
   export CLASSPATH *****
   ```

   or, depending on the type of Linux shell you use:

   ```
   setenv CLASSPATH *****
   ```

   You only need to do this once per session.

   Most systems offer ways of setting environment variables other than from the command line; e.g. in Windows XP this can be done in:

   `Control Panel` → `System` → `Advanced` → `Environment Variables`

   The advantage of the latter method is that these settings are persistent (i.e. you do not have to set the class path each time you want to use the system again).

In any case the `*****` has to be replaced by a list of (absolute or relative) paths to directories and/or `jar` files. On a Windows system this list is separated by semicolons, e.g.

```
%classpath%;c:\java\libraries\;..\KUL_JCHR.jar;.
```

On a Linux system the list is separated by colons:

```
$CLASSPATH:/home/java/libraries/:../KUL_JCHR.jar:.
```

Note that we included the current working directory – as is necessary most of the time – in both examples by including '`.`'. Also, we included the current classpath as well. Leaving this first item out of the list, allows you to overwrite the classpath environment variable rather than extending it. Since in general using the K.U.Leuven JCHR System requires several libraries (i.e. jar files), the second option (the environment variable) seems preferable. It is generally easier to include all jars directly in your class search path, even if some are not strictly needed. For details on the required libraries, we refer to sections 5.1 and 6.1.

Appendix C

Benchmarking JCHR

The K.U.Leuven JCHR website [VW06] contains a series of benchmarks, of which some results are shown in [VW05, VWSD05]. The Prolog-counterparts of these benchmarks can be found on the website of the K.U.Leuven CHR System [Sch06].

Bibliography

[AKSS01] Slim Abdennadher, Ekkerhard Krämer, Matthias Saft, and Matthias Schmauß. JACK: A Java Constraint Kit. In Michael Hanus, editor, *Proceedings of the International Workshop on Functional and (Constraint) Logic Programming WFLP*, volume 64 of *Electronic Notes in Theoretical Computer Science*, pages 1–17, Kiel, Germany, September 13–15 2001.

[Bra04] Gilad Bracha. *Generics in the Java Programming Language*. Sun Microsystems, Inc., July 2004. `http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf`.

[BV94] Jacques Bouaud and Robert Voyer. Behavioral match: Embedding production systems and objects. In Pachet [eoo94].

[dFFR00] Carlos Santos da Figueira Filho and Geber Lisboa Ramalho. JEOPS - The Java Embedded Object Production System. In Maria Monard and Jaime S Sichman, editors, *Advances in Artificial Intelligence: Proceedings of International Joint Conference, 7th Ibero-American Conference on AI, 15th Brazilian Symposium on AI (IBERAMIA-SBIA)*, volume 1952 of *Lecture Notes in Computer Science (LNCS) – Lecture Notes in Artificial Intelligence (LNAI)*, pages 53–62, Atibaia, Brazil, 19–22 November 2000.

[dlBDMS02] María García de la Banda, Bart Demoen, Kim Mariott, and Peter J. Stuckey. To the gates of HAL: a HAL tutorial. In Zhenjiang Hu and M. Rodrguez-Artalejo, editors, *Proceedings of the Sixth International Symposium on Functional and Logic Programming*, number 2441 in Lecture Notes in Computer Science, pages 47–66, Aizu, Japan, September 15–17 2002. Springer-Verlag.

[DSdlBH03] Gregory J. Duck, Peter J. Stuckey, María García de la Banda, and Christian Holzbaur. Extending Arbitrary Solvers with Constraint Handling Rules. In Dale Miller, editor, *Proceedings of the Fifth ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, pages 79–90, Uppsala, Sweden, August 27–29 2003. ACM Press.

[DSdlBH04] Gregory J. Duck, Peter J. Stuckey, María García de la Banda, and Christian Holzbaur. The refined operational semantics of Constraint Handling Rules. In Bart Demoen and Vladimir Lifschitz, editors, *Proceedings of the 20th International Conference on Logic Programming (ICLP)*, volume 3132 of *Lecture Notes in Computer Science (LNCS)*, pages 90–104. Springer-Verlag, September 2004.

[Duc05] Gregory Duck. Haskell CHR. `http://www.cs.mu.oz.au/~gjd/haskellchr/`, March 2005.

[eoo94] Proceedings of the OOPSLA'94 workshop on Embedded Object-Oriented Production Systems (EOOPS). In F. Pachet, editor, *Proceedings of the OOPSLA'94 workshop on Embedded Object-Oriented Production Systems (EOOPS)*, Portland, Oregon, October 1994.

[FB95a] Thom Frühwirth and P. Brisset. *ECL^{i}PS^e 3.5.1 Extensions User Manual*, chapter Chapter on Constraint Handling Rules. ECRC, Munich, Germany, December 1995.

[FB95b] Thom Frühwirth and Pascal Brisset. High-level implementations of Constraint Handling Rules. Technical Report ECRC-95-20, ECRC, Munich, Germany, June 1995.

[For82] Charles L. Forgy. Rete: a fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19:17–37, 1982.

[fre06] FreeMarker Template Engine. `http://www.freemarker.org/`, August 2006.

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

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

[GJSB04] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison-Wesley Professional, third edition, 2004. Available online at `http://java.sun.com/docs/books/jls/`.

[H$^+$06] Christian Holzbaur et al. *SICStus Prolog Manual*, chapter Constraint Handling Rules. 3.12.5 edition, March 2006.

[HdlBSD05] Christian Holzbaur, María García de la Banda, Peter J. Stuckey, and Gregory J. Duck. Optimizing Compilation of Constraint Handling Rules in HAL. *Theory and Practice of Logic Programming – Special Issue on Constraint Handling Rules*, 5(4–5):503–531, July/September 2005.

[HF99] Christian Holzbaur and Thom Frühwirth. Compiling constraint handling rules into Prolog with attributed variables. In G. Nadathur, editor, *Proceedings of the International Conference on Principles and Practice of Declarative Programming*, number 1702 in Lecture Notes in Computer Science, pages 117–133. Springer Verlag, 1999.

[HF00] Christian Holzbaur and Thom Frühwirth. A Prolog Constraint Handling Rules Compiler and Runtime System. *Journal of Applied Artificial Intelligence, Special Issue on Constraint Handling Rules*, 14(4):369–388, April 2000.

[Kaw06] Kohsuke Kawaguchi. args4j – Java command line option parsing library. `https://args4j.dev.java.net/`, September 2006.

[MBLG90] Daniel P. Miranker, David A. Brant, Bernie Lofaso, and David Gadbois. On the Performance of Lazy Matching in Production Systems. In *8th National Conference on Artificial Intelligence*, pages 685–692. AAAI, July 1990.

[P+06] Terence Parr et al. ANTLR Parser Generator. `http://www.antlr.org/`, August 2006.

[Pac95] François Pachet. On the Embeddability of Production Rules in Object-Oriented Languages. *Journal of Object-Oriented Programming (JOOP)*, 8(4):19–24, July/August 1995.

[Sch99] Matthias Schmauß. An Implementation of CHR in Java. Diplomarbeit (german msc thesis), Department of Computer Science, University of Munich, Germany, 1999.

[Sch05] Tom Schrijvers. *Analyses, optimizations and extensions of Constraint Handling Rules*. PhD thesis, K.U.Leuven, Leuven, Belgium, June 2005.

[Sch06] Tom Schrijvers. The K.U.Leuven CHR System. `http://www.cs.kuleuven.ac.be/~toms/Research/CHR/`, August 2006.

[SD04] Tom Schrijvers and Bart Demoen. The K.U.Leuven CHR system: implementation and application. In Thom Frühwirth and Marc Meister, editors, *First Workshop on Constraint Handling Rules: Selected Contributions*, Ulm, Germany, May 2004.

[SDD+05a] Tom Schrijvers, Bart Demoen, Gregory J. Duck, Peter J. Stuckey, and Thom Frühwirth. Automatic implication checking for CHR constraint solvers. Technical Report CW 402, K.U.Leuven, Department of Computer Science, Leuven, Belgium, January 2005.

[SDD+05b] Tom Schrijvers, Bart Demoen, Gregory J. Duck, Peter J. Stuckey, and Thom Frühwirth. Automatic implication checking for CHR constraints. In Horatiu Cirstea and Narciso Marti-Oliet, editors, *Proceedings of the 6th International Workshop on Rule-Based Programming*, pages 93–111, Nara, Japan, April 23 2005.

[SF+06] Tom Schrijvers, Thom Frühwirth, et al. CHR Homepage. `http://www.cs.kuleuven.be/~dtai/projects/CHR`, August 2006.

[Smi01] Chris Smith. Consistent comparisons. `http://cdsmith.twu.net/professional/java/pontifications/comparison.html`, June 6 2001.

[SS01] Peter J. Stuckey and Martin Sulzmann. A systematic approach in type system design based on Constraint Handling Rules. In *Third Workshop on Rule-Based Constraint Reasoning and Programming*, December 2001.

[SS05] Peter J. Stuckey and Martin Sulzmann. A theory of overloading. *ACM Trans. Program. Lang. Syst.*, 27(6):1216–1269, 2005.

[SSD05a] Jon Sneyers, Tom Schrijvers, and Bart Demoen. Guard and Continuation Optimization for Occurrence Representations of CHR. In Maurizio Gabbrielli and Gopal Gupta, editors, *Proceedings of the 21st International Conference on Logic Programming (ICLP)*, volume 3668 of *Lecture Notes in Computer Science (LNCS)*, pages 83–97, October 2005.

[SSD05b] Jon Sneyers, Tom Schrijvers, and Bart Demoen. Guard and Continuation Optimization for Occurrence Representations of CHR. Technical Report CW 420, K.U.Leuven, Department of Computer Science, Leuven, Belgium, July 2005.

[Suna] Sun Microsystems, Inc. Code conventions for the Java programming language. `http://java.sun.com/docs/codeconv/`.

[Sunb] Sun Microsystems, Inc. Java Homepage. `http://java.sun.com/`.

[Sun05] Sun Microsystems, Inc. Java bug #2125378. `http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=2125378`, April 14 2005.

[Sun06a] Sun Microsystems, Inc. Java Beans. `http://java.sun.com/products/javabeans/`, August 2006.

[Sun06b] Sun Microsystems, Inc. The Java Tutorials – JavaBeans – Lesson: Properties. `http://java.sun.com/docs/books/tutorial/javabeans/properties/`, August 2006.

[Sun06c] Sun Microsystems, Inc. JavaDoc Tool Home Page. `http://java.sun.com/j2se/javadoc/`, August 2006.

[SWD03] Tom Schrijvers, David Warren, and Bart Demoen. CHR for XSB. In R. Lopes and M. Ferreira, editors, *Proceedings of CICLOPS 2003: Colloquium on Implementation of Constraint and LOgic Programming Systems*, pages 7–20, 2003.

[SWD05] Tom Schrijvers, Jan Wielemaker, and Bart Demoen. Constraint handling rules for swi-prolog. In *9th Workshop on (Constraint) Logic Programming*, Ulm, February 2005. Abstract Only.

[SZD06] Tom Schrijvers, Neng-Fa Zhou, and Bart Demoen. Translating Constraint Handling Rules into Action Rules. In Tom Schrijvers and Thom Frühwirth, editors, *Proceedings of the Third Workshop on Constraint Handling Rules (CHR 2006)*, Venice, Italy, 9 July 2006.

[tur]

[VA06] Jairson Vitorino and Marcus Aurelio. CHORD. `http://sourceforge.net/projects/chord/`, August 2006.

[VW05] Peter Van Weert. Constraint Programming in Java: een gebruiksvriendelijk, flexibel en efficient CHR-Systeem voor Java. Master's thesis, K.U.Leuven, Belgium, May 2005.

[VW06] Peter Van Weert. JCHR Homepage. `http://www.cs.kuleuven.be/~petervw/JCHR/`, August 2006.

[VWSD05] Peter Van Weert, Tom Schrijvers, and Bart Demoen. K.U.Leuven JCHR: a user-friendly, flexible and efficient CHR system for Java. In Tom Schrijvers and Thom Frühwirth, editors, *Proceedings of the Second Workshop on Constraint Handling Rules (CHR 2005)*, pages 47–62, Sitges, Spain, 5 October 2005.

[Wol01] Armin Wolf. Adaptive constraint handling with CHR in Java. In Toby Walsh, editor, *Proceedings of the 7th International Conference on Principles and Practice of Constraint Programming (CP 2001)*, volume 2239 of *Lecture Notes in Computer Science (LNCS)*, pages 256–270, Paphos, Cyprus, November 26–December 1 2001. Springer.

List of Acronyms

ANTLR	ANother Tool for Language Recognition
API	Application Program Interface
CHR	Constraint Handling Rules
CP	Constraint Programming
C(L)P	Constraint (Logic) Programming
DJCHR	Dynamic Java Constraint Handling Rules
ECL^{i}PS^e	ECRC Constraint Logic Parallel System (cf. ECL^{i}PS^e-Prolog)
ECRC	European Computer-Industry Research Centre
HAL	Heuristically programmed ALgorithmic computer (after the "HAL 9000" from the Stanley Kubrick and Arthur C. Clarke movie *2001: A Space Odyssey*)
HTML	HyperText Markup Language
IDE	Integrated Development Environment
J2SE	Java 2 Standard Edition
JaCK	Java Constraint Kit
jar	Java ARchive
JCHR	Java Constraint Handling Rules
JDK	J2SE Development Kit
JRE	J2SE Runtime Environment
K.U.Leuven	Katholieke Universiteit Leuven (Catholic University Leuven)
LEAPS	Lazy Evaluation Algorithm for Production Systems
Prolog	Programming in Logic
SICS	Swedish Institute of Computer Science
SDK	Software Development Kit
SWI	Sociaal-Wetenschappelijke Informatica (cf. SWI-Prolog)