A Type-Safe Embedding of Constraint Handling Rules into Haskell

Wei-Ngan Chin, Martin Sulzmann and Meng Wang

School of Computing, National University of Singapore
S16 Level 5, 3 Science Drive 2, Singapore 117543

Abstract. Two of the most exciting developments in programming languages research over the last decade have been constraint programming on the one hand, and typeful functional programming on the other hand. Constraint programming (CP) is an emergent software technology for declarative description and effective solving of large combinatorial problems, especially in the areas of planning and scheduling. Based on a strong theoretical foundation, this technology is also attracting widespread commercial interests. One key reason for its popularity is that constraints allow end-users to directly formalize the dependencies of the physical worlds; and through sound mathematical abstractions we can have the computers to understand and solve these dependencies. Higher-level programming languages have come a long way. One of the most promising technologies is advanced functional languages with a rich set of versatile features, such as higher-order functions, type generics, and algebraic structures. These features allow high-level language to function as better modular glues for constructing larger and yet safer software, providing precision and reusability to programming. In this thesis, we discuss how a constraint-solving technology, called Constraint-Handling Rules (CHR), can be embedded as an extension of a mainstream functional programming language, known as Haskell, in a type-safe fashion. We present various techniques used in the embedding which produces an excellent tool equipped with power from both worlds.

Keywords. Constraint Handling Rule, Haskell, Type System, Embedding, Monad

1 Introduction

Two of the most exciting developments in programming languages research over the last decade have been constraint programming [8] on the one hand, and typeful functional programming [11] on the other hand.

Constraint programming (CP) is an emergent software technology for declarative description and effective solving of large combinatorial problems, especially in the areas of planning and scheduling. Based on a strong theoretical foundation, this technology is also attracting widespread commercial interests. One key reason for its popularity is that constraints allow end-users to directly formalize the dependencies of the physical worlds; and through sound mathematical abstractions we can have the computers to understand and solve these dependencies.

Constraint Handling Rules (CHR) [3] is a newly proposed high-level language extension especially designed for writing constraint solvers. With CHR, one can introduce user-defined constraints into a given host language. As a language extension, CHR are only concerned with constraints, all auxiliary computations are performed directly in the host language. Compare to other programming languages in constraint programming family, CHR is more general since it allows "multiple heads", i.e. conjunctions of constraints in the heard of a rule. With multi-headed CHR, more sophisticated constraint systems could be specified in a natural manner.

In another front, higher-level programming languages have come a long way. One of the most promising technologies is advanced functional languages with a rich set of versatile features, such as higher-order functions, type generics, and algebraic structures. These features allow high-level language to function as better modular glues for constructing larger and yet safer software, providing precision and reusability to programming. A desirable feature of programming languages is safety. Broadly speaking, safe programming languages capture a significant class of errors that can be detected at compile-time; thus adding to the robustness of well-typed software. Statically typed languages, like Haskell [9], do all the typechecking at compile time and type information is discarded after compilation. As a result, it is usually not possible to write functions whose behaviour depends on run-time type information.

Many proposals have been made for the integration of the two programming paradigms. Among these, the most well-known ones are the functional logic programming languages Curry [5], Escher [10], and Mercury [12].

Curry and Escher can be seen as variations on Haskell, where constraint logic programming features are added. Mercury can be seen as an improvement of Prolog, where types and functional programming features are added. All these are completely new and autonomous languages.

These languages demonstrate the impressive power gained from offering useful features of both worlds; to achieve this additional expressivity they have to adopt rather complicated semantics. In addition, great amount of effort is required for the developer to build a new compiler, and for the user to learn a new language.

An alternative approach which has gained a lot of popularity is to *embed* [7] a new language in another language, called the *host language*; the host language with these added functionality become the new language. As a consequence,

the new language comes equipped with all the features of the host language, with no additional work on the part of the language designer. Embedding works particularly well when the host language is a functional language.

The embedding approach has another important advantage. Programs in the embedded language are first class citizens in the host language, can therefore be generated by a program in the host language.

Our work is in line with the second approach. The aim is to embed CHR into Haskell. There are a few existing implementations of CHR. The most well-known one is implemented in Prolog [4]. This implementation enjoys great support from the host language, Prolog. Issues like logical variables are handled trivially. Multi-head CHR rules are transformed into single-headed Prolog clauses. However, this CHR implementation is completely untyped, follows the host language.

A more recent implementation of CHR , JACK [13], is in Java. The CHR programs are compiled into Java code which is intended to be integrated into Java applications or applets. Though it provids a powerful tool, the syntax of the language is very heavy as a result of imperative embedding.

There was also an early CHR implementation in Lisp. Unfortunately, no document or code remains today.

Closely related to this project is a full CHR solver that was built for a more advanced type checking in Haskell, called Chameleon [14]. This implementation provides the foundation of the core CHR solver used in this project. Chameleon was designed as a stand alone system catered for specific applications. It is not integrated with the host language. Moreover, the CHR declarations are not typed.

In this project, we achieved a type-safe embedding of CHR into Haskell. We developed a monomorphic type system for CHR base on which we perform type checking for CHR declarations. We explored various techniques in building generic term representation which allows us to refer to Haskell types within CHR declarations. We also employed program generation techniques to automatically generate generic unification. Some preprocessing are used to provide elegant syntax. The resulting embedding language which we call HCHR, is extensible, expressive, user friendly, and type safe.

We begin by looking at a small motivating example. Consider we have the following Haskell data type which describes employees in a university.

Example 1.1.

$$\textbf{data}\ \mathit{Employee} = \mathit{Academic}\ \mathit{String}\ \mathit{Int} \mid \mathit{Nonacademic}\ \mathit{String}$$

Now suppose we want to solve some constraints that make use this data type. If a stand alone CHR solver is used, we firstly need to interface to the solver and be able to model this data type in the solver's syntax. After the solving is completed, we also have to find some way to query the result of the solving and convert it back to Haskell types in order to use them.

Is there a better way to do this? Our solution is to integrate a CHR solver with Haskell. In this new setting, we can define predicates and constraints involving Haskell data types as follows:

$$\textbf{data}\ \mathit{Pred} = C\ (\mathit{Int}, \mathit{Employee}, \mathit{Bool})$$

rule $C\ (x,\ Academic\ \text{“}Loser\text{”}\ y\ ,\ True)\ \Leftrightarrow\ x = y\ ;$
rule $C\ (x,\ Academic\ \text{“}Winner\text{”}\ y,\ z)\ \Rightarrow\ x = y\ , z = False\ ;$

A predicate C and two CHR rules are defined in the above code fragment.

In addition, we also define a solve and a query function which invokes the CHR solver and queries the result respectively:

$$
\begin{array}{l}
run\ =\ runM\ (\mathbf{do} \\
\qquad\qquad\quad \vdots \\
\qquad\qquad\quad solve\ aGoal \\
\qquad\qquad\quad r\ \leftarrow\ (query\ \text{“}x\text{”}) \\
\qquad\qquad\quad f\ r)
\end{array}
$$

where *aGoal* is defined as:

$$aGoal\ =\ C\ (5,\ Academic\ \text{“}Winner\text{”}\ x,\ z)$$

In this piece of code, the programmer calls the CHR solver within this Haskell function by passing a goal named *aGoal* as its argument. After that, the value of variable x in the goal is queried and passed to another function call.

We can see from the this example, Haskell programmers are totally relieved from the interfacing and conversion required to invoke a constraint solver from Haskell. He is able to program completely in Haskell will a little syntax extension. This provides programmers expressiveness from both programming paradigms with minimal cost.

We begin in section 2 where we briefly introduce the syntax and operational semantics of CHR. In section 3, we present the type system of CHR and how the type checking is done practically. In section 4, various embedding and code generation techniques are discussed. After that, We show the implementation issues of the CHR solver and the interaction between CHR solver and Haskell. In section 6, we give some applications using HCHR. Lastly, we conclude with directions of future works.

2 Preliminaries

In this chapter we give the syntax and an operational semantics of Constraint Handling Rules which are adopt from [1] with minor modifications.

Constraints are considered to be special first-order predicates. We use two disjoint sorts of predicate symbols: *built-in predicates* and *user-defined predicates*. Intuitively, built-in predicates are defined by some constraint theory and handled by an appropriate constraint solver, while user-defined predicates are those defined by a CHR program. We call an atomic formula with a built-in predicate a built-in constraint and atomic formula with a user-defined predicated a user-defined constraint.

2.1 Syntax of CHR

Definition 1. *A* CHR *program is a finite set of rules. There are two basic kinds of rules. A* simplification rule *is of the form*

$$H_1,\cdots,H_m \Leftrightarrow G_1,\cdots,G_n \mid B_1,\cdots,B_i$$

A propagation rule *is of the form*

$$H_1,\cdots,H_m \Rightarrow G_1,\cdots,G_n \mid B_1,\cdots,B_i$$

where the head $H_1,\cdots,H_m$ is a nonempty conjunction of user-defined constraints. The guard $G_1,\cdots,G_n$ is a conjunction of built-in constraints, while the body $B_1,\cdots,B_i$ is a conjunction of built-in and user-defined constraints. Conjunctions of constraints in the body are called goals. *If the guard is empty, the symbol | is omitted.*

2.2 Operational Semantics of CHR

The *operational semantics* of CHR programs is given by a transition system.

Definition 2. *A* state *is a tuple*

$$< \mathcal{G}_s,\mathcal{C}_U,\mathcal{C}_B,\mathcal{T},\mathcal{V} >.$$

$\mathcal{G}_s$ is a conjunction of user-defined and built-in constraints called goal store. *$\mathcal{C}_U$ is a conjunction of user-defined constraints, likewise $\mathcal{C}_B$ is a conjunction of built-in constraints. $\mathcal{C}_U$ and $\mathcal{C}_B$ are called* user-defined and built-in (constraint) store, *respectively. $\mathcal{T}$ is a set of tokens (token store) of the form $\mathcal{R}@\mathcal{C}$, where $\mathcal{C}$ is a conjunction of user-defined constraints and $\mathcal{R}$ a rulename. An empty goal store or user-defined store is represented by $\top$. The built-in store cannot be empty. In its most simple form, it consists only of* True *or* False. *$\mathcal{V}$ contains all the variables in the goal introduced from outside when the solver initially starts. It does not change during the computation. The variables in $\mathcal{V}$ are called* global.

Intuitively, $\mathcal{G}_S$ contains the constraints that remain to be solved, $\mathcal{G}_B$ and $\mathcal{C}_U$ are the built-in and the user-defined constraints, respectively, accumulated and simplified so far. $\mathcal{T}$ contains information about the propagation rules with the respective constraints which they can be possibly applied on.

Computation Steps Given a CHR program P we define the transition relation $\mapsto$ by introducing four kinds of *computation steps*. The aim of the computation is to incrementally reduce arbitrary states to states that contain no goals and, if possible, the simplest form of user-defined constraints.

Solve

$$\frac{\mathcal{C} \text{ is a built-in constraint}}{< \mathcal{C} \wedge \mathcal{G}_S, \mathcal{C}_U, \mathcal{C}_B, \mathcal{T}, \mathcal{V} > \mapsto < \mathcal{G}_S, \mathcal{C}_U, \mathcal{C} \wedge \mathcal{C}_B, \mathcal{T}, \mathcal{V} >}$$

Introduce

$$\frac{\mathcal{C} \text{ is a user-defined constraint}}{< \mathcal{C} \wedge \mathcal{G}_S, \mathcal{C}_U, \mathcal{C}_B, \mathcal{T}, \mathcal{V} > \mapsto < \mathcal{G}_S, \mathcal{C} \wedge \mathcal{C}_U, \mathcal{C}_B, \mathcal{T}, \mathcal{V} >}$$

Simplify

$(\mathcal{H} \Leftrightarrow \mathcal{G} \mid \mathcal{B})$ is a fresh variant of a rule in P with the variables $\bar{a}$

$$\frac{CT \models \mathcal{C}_B \rightarrow \exists \bar{x}(\mathcal{H} \doteq \mathcal{H}' \wedge \mathcal{G})}{< \mathcal{G}_S, \mathcal{H}' \wedge \mathcal{C}_U, \mathcal{C}_B, \mathcal{T}, \mathcal{V} > \mapsto < \mathcal{G}_S \wedge \mathcal{B}, \mathcal{C}_U, \mathcal{C}_B, \mathcal{T}, \mathcal{V} >}$$

Propagate

$(\mathcal{R}@\mathcal{H} \Leftrightarrow \mathcal{G} \mid \mathcal{B})$ is a fresh variant of a rule in P with the variables $\bar{a}$

$$\frac{CT \models \mathcal{C}_B \rightarrow \exists \bar{x}(\mathcal{H} \doteq \mathcal{H}' \wedge \mathcal{G}) \qquad \{\mathcal{R}@\mathcal{H}\} \notin \mathcal{T}}{< \mathcal{G}_S, \mathcal{H}' \wedge \mathcal{C}_U, \mathcal{C}_B, \mathcal{T}, \mathcal{V} > \mapsto < \mathcal{G}_S \wedge \mathcal{B}, \mathcal{H}' \wedge \mathcal{C}_U, \mathcal{H} \doteq \mathcal{H}' \wedge \mathcal{C}_B, \{\mathcal{R}@\mathcal{H}\} \cup \mathcal{T}, \mathcal{V} >}$$

$\doteq$ denotes syntactic equality in the above steps. In the **Solve** computation step, the built-in solver moves a constraint $\mathcal{C}$ from the goal store to the built-in store. **Introduce** transports a user-defined constraint $\mathcal{C}$ from the goal store into the user-defined constraint store. To **Simplify** user-defined constraints $\mathcal{H}'$ means to replace them by the body $\mathcal{B}$ of a fresh variant[1] of a simplification rule $(\mathcal{H} \Leftrightarrow \mathcal{G} \mid \mathcal{B})$ from the program, provided $\mathcal{H}$' matches[2] the head $\mathcal{H}$ and the resulting guard $\mathcal{G}$ is implied by the built-in constraint store. To **Propagate** user-defined constraints $\mathcal{H}'$ means to add $\mathcal{B}$ to the goal store $\mathcal{G}_S$ and add the token $\mathcal{R}@\mathcal{H}$ to the token store if $\mathcal{H}'$ matches the head $\mathcal{H}$ of a propagation rule $(\mathcal{H} \Rightarrow \mathcal{G} \mid \mathcal{B})$ in the program and the resulting guard $\mathcal{G}$ is implied by the built-in constraint store.

We require that the rules are applied fairly, i.e. that every rule that is applicable is applied eventually. Fairness is respected and trivial non-termination is avoided by applying a propagation rule at most once to the same constrains.

Token Store A propagation rule needs an applicability condition to avoid trivial nontermination. Our approach stores information about propagation rules which has been applied to a given set of user-defined constraints. Thus it cannot be reapplied to the set of user-defined constraints. Once a propagation rule has been applied to user-defined constraints, the appropriate token is added to the token store.

Definition 3. *A* initial state *for a goal $\mathcal{G}_s$ is of the form:*

$$< \mathcal{G}_s, \top, true, \phi, \mathcal{V} >.$$

where $\mathcal{V}$ is the set of variables occurring in $< \mathcal{G}_s$.

A final state *is either of the form*

$$< \mathcal{G}_s, \mathcal{C}_U, false, \mathcal{T}, \mathcal{V} >.$$

(such a state is called failed*) or of the from*

$$< \top, \mathcal{C}_U, \mathcal{C}_B, \mathcal{T}, \mathcal{V} >.$$

with no computation step possible anymore and $\mathcal{C}_B$ is not false *(such a state is called* successful*).*

[1] Two expressions are variants if they can be obtained from each other by a variable renaming. A fresh variant contains only variables that do not occur in the state.

[2] Matching rather than unification is the effect of the existential quantification over the head equalities: $\exists \bar{x}(\mathcal{H} \doteq \mathcal{H}')$.

3 Typed CHR

A desirable feature of programming languages is safety. Broadly speaking, safe programming languages capture a significant class of errors that can be detected at compile-time. *Static Type Checking* is a very effective way to enforce safety. Thus we would like to introduce this feature in our embedded CHR.

3.1 CHR Type System

Terms The *Terms*, denoted as t, in our system include logical variable, constant, data type, and function application.

$$t ::= x \mid n \mid C\ \bar{t} \mid (f\ t)$$

Constraints The constraints in our system include user-defined constraints and HERBRAND built-in constraints.

$$C ::= U_c\ \bar{t} \mid t_1 = t_2$$

where $\bar{t}$ is a stream of terms of the form $t_1 \ldots t_n$.

Types Let us first introduce the notions of *types* and *type environments*:

$\tau \in$ **Type** types
$\Gamma \in$ **TEnv** type environments

The types in our system are given by

$$\tau ::= \alpha \mid \tau_1 \rightarrow \tau_1 \mid D\ \bar{\tau} \mid \tau_{pred} \mid \tau_{CHR}$$

where α denotes base types and as usual we use arrow for function types. D is a data type constructor which take in a type stream $\bar{\tau}$. There are two special types τ_{pred} and τ_{CHR} which represents the type of predicates and CHR rules.

Type Rules The initial type environment is: $\Gamma_{init} = \{U_c :: \bar{\tau} \rightarrow \tau_{pred}\}$ where type information of all the user-defined predicates are present.

The general form of a typing is given by

$$\Gamma \vdash e : \tau$$

that says that the expression e has type τ assuming that any free variable has type given by Γ. The axioms and rules for the judgements are listed and explained below.

$$[con] \quad \Gamma_{init} \vdash c : \tau_c$$

$$[var] \quad \Gamma_{init} \vdash x : \tau \quad \text{if } \Gamma_{init}(x) = \tau$$

$$[eq] \quad \frac{\Gamma_{init} \vdash e_1 : \tau \quad \Gamma_{init} \vdash e_2 : \tau}{\Gamma_{init} \vdash e_1 = e_2 : \tau_{pred}}$$

$$[app] \quad \frac{\Gamma_{init} \vdash e_1 : \tau_2 \rightarrow \tau_0 \quad \Gamma_{init} \vdash e_2 : \tau_2}{\Gamma_{init} \vdash e_1\ e_2 : \tau_0}$$

$$[prop] \quad \frac{\Gamma_{init} \vdash h_i : \tau_{pred} \quad \Gamma_{init} \vdash g_j : \tau_{pred} \quad \Gamma_{init} \vdash b_k : \tau_{pred}}{\Gamma_{init} \vdash h_1 \ldots h_m \Rightarrow g_1 \ldots g_n \mid b_1 \ldots b_o : \tau_{CHR}} \text{ where } i = \{1, \ldots, m\} \wedge j = \{1, \ldots, n\} \wedge k = \{1, \ldots, o\}$$

$$[simp] \quad \frac{\Gamma_{init} \vdash h_i : \tau_{pred} \quad \Gamma_{init} \vdash g_j : \tau_{pred} \quad \Gamma_{init} \vdash b_k : \tau_{pred}}{\Gamma_{init} \vdash h_1 \ldots h_m \Leftrightarrow g_1 \ldots g_n \mid b_1 \ldots b_o : \tau_{CHR}} \text{ where } i = \{1, \ldots, m\} \wedge j = \{1, \ldots, n\} \wedge k = \{1, \ldots, o\}$$

The axioms [*con*] and [*var*] are straightforward: the first uses the predefined type for the constant and the second consults the type environment. In the rule [*eq*], we require expressions on both sides of = must have the same type. The resulting type is τ_{pred}. The rule [*app*] requires that the operator and the operand of the application can be typed and implicitly it requires that the type of the operator is a function type where the type before the arrow equals that of the operand - in this way we express that the types of the formal and actual parameter must be equal. The rules [*prop*] and [*simp*] are straight forward. The head, guard, and body of the rules are all of type τ_{pred} and the CHR consists of them is of type τ_{CHR}.

3.2 Type Checking

Based on the typing rules defined in the previous section, we can type check the CHR declarations. One approach is to write a type checking program which is rather straight forward for a monomorphic type system. However, this approach does not scale to more sophisticated types. If one wishes to extend the simply typed CHR to a powerful polymorphic one, as in Haskell, then the type checking can be hard to implement.

To avoid reinventing the wheel, we propose an approach that make use of Haskell type checker to perform the type checking for CHR. Specifically, we will generate a special type checking code from each HCHR program. The type checking code is equivalent to the CHR declarations of type correctness. A transformation from the CHR syntax to its corresponding type checking code is defined. This transformation translates CHR declarations to approximated λ abstraction statements that reflect the typing constraints.

Definition 4. *A* type checking program*(TC) is a tuple*

$$< \mathcal{F}, \mathcal{A} >.$$

$\mathcal{F}$ and $\mathcal{A}$ correspond to a set of λ abstractions and a set of auxiliary Haskell functions, respectively.

Definition 5. *A transformation $\mathcal{TR}$: CHR $\rightsquigarrow$ TC, where TC denotes type checking code for the CHR, is done in two steps.*

Let $\mathcal{TR}(R)$=$< \mathcal{F}, \mathcal{A} >$.

1. Auxiliary function creation:*A Haskell function eq is created to ensure both terms under the equality constraints are of the same type. See the use of it in step 2.*

   $\mathcal{A}$=

   $eq :: (\alpha \rightarrow \alpha \rightarrow \beta)$

   $eq = undefined$ — *undefined* is a Haskell function which has arbitrary type.

2. CHR transformation: *Simplification and Propagation rules from R:*

   $$UC_1,\cdots,UC_m \Leftrightarrow BC_1,\cdots,BC_n \mid UC'_1,\cdots,UC'_i,BC'_1,\cdots,BC'_j$$

   $$UC_1,\cdots,UC_m \Rightarrow BC_1,\cdots,BC_n \mid UC'_1,\cdots,UC'_i,BC'_1,\cdots,BC'_j$$

   where heads and guards of the rules are represented as a set of user-defined constraints and built-in constraints respectively. Bodies of the rules is a set consists of both user-defined and built-in constraints. The rules are transformed to the following form (denoted by $\mathcal{F}$):

   $$\lambda\ \bar{v} \rightarrow (UC_1,\cdots,UC_m,EQ_1,\cdots,EQ_n,UC'_1,\cdots,UC'_i,EQ'_1,\cdots,EQ'_j)$$

   where

   $\bar{v}$=*All the logical variables are present in* $UC_1,\cdots,UC_m$.

   which each $EQ_k = eq\ t_{k1}\ t_{k2}$ *comes from* BC_k *that is of the form* $t_{k1} = t_{k2}$.

An important property of $\mathcal{TR}$ is that it preserves the type correctness of CHR declarations.

Definition 6. *Let R be a CHR rule*
Then the following equivalence holds

$$\Gamma_{init} \vdash R : \tau_{CHR} \text{ iff } \mathcal{TR}(R) \text{ is type correct.}$$

Let us consider the employee example with small modifications.

Example 1.

data $Employee\ =\ Academic\ String\ Int \mid Nonacademic\ String$

data $Pred\ =\ C\ (Int, Employee, Bool)$

rule $C\ (x,\ Academic\ \text{“}Loser\text{”}\ y\ , 3)\ \Leftrightarrow\ x = y$

rule $C\ (x, Academic\ \text{“}Winner\text{”}\ y,\ z)\ \Rightarrow x = z, z = False$

These two CHRs are not type correct. In the first rule, a *Int* value 3 is used in the place of a *Bool*. In the second rule, the logical variable z is used as an *Bool* in LHS and RHS. This is obviously a type error. Now we show how the transformation is done:

First, an auxiliary function *eq* is created:

$eq :: (\alpha \rightarrow \alpha \rightarrow \beta)$
$eq = undefined$

Second, all the logic variables in the rules become formal parameters of the functional abstractions to enforce that all occurrences of the same variable must have the same type

$r1 = (\lambda\,(x\ y)\ \rightarrow\ ([C\ (x, Academic\ \text{“}Loser\text{”}\ y, 3)], [],\ [(eq\ x\ y)]))$
$r2 = (\lambda\,(x\ y\ z)\ \rightarrow\ ([(C\ (x, Academic\ \text{“}Winner\text{”}\ y, z))], [],\ [(eq\ x\ z), (eq\ z\ False)]))$

Note that the function *eq* ensures the terms on both side of the equality constraints have the same type.

After this transformation, the generated type checking code is passed to the Haskell compiler. The Haskell type checker will be able to detect any type discrepancies in the original HCHR program[3].

The error message generated in GHCi(a widely used Haskell interpreter.) for this example is given below:

```
TC.hs:8:

    Couldn't match 'Bool' against 'Int'

        Expected type: Bool

        Inferred type: Int

    When checking the type signature of the expression: 3 :: Int

    In the first argument of 'C', namely

        '(x, Academic "Loser" y, (3 :: Int))'


TC.hs:9:
```

[3] We focus on the type checking of the CHR program because the Haskell part of HCHR programs are checked by Haskell compiler without additional effort.

```
    Couldn't match 'Int' against 'Bool'

        Expected type: Int

        Inferred type: Bool

    In the second argument of 'eq', namely 'z'

    In the list element: (eq x z)

Failed, modules loaded: none.
```

Note that the above message gives a clear indication that the error is caused by trying to unify a *Int* value with a *Bool* value. Additional power of type checking can be gained from using more advanced Haskell type checker. One good example is *Chameleon* [14] which gives more informative error message compare to native Haskell type checker especially in the presence of polymorphism and overloading.

In a real implementation, the type checking code is appended at the end of the internal solver code. Such a type checking code section does nothing but to report compilation errors if the original CHR code is not well typed.

4 The Embedding

Before we can implement the CHR solver, there are some essential data structures and operations that must be provided. I will discuss how to tackle them before going into the implementation of the core solver.

4.1 Logical Variable and Term

When talking about embedding a constraint logic programming language into a functional language. The first question asked is: How to represent logical variables? This has been shown to be non-trivial. In Hinze's attempt to embed Prolog's control constructs in Haskell [6], he simply chose not to support logical variables because logical variables "do not go well with the use of Haskell as a host language". A previous embedding of CHR in Haskell done by Sulzmann [14] addressed this issue by using a simple approach through the us of *universal type*, namely:

$$\textbf{data}\ \mathit{Term} = \mathit{Var}\ \mathit{Id} \mid \mathit{Const}\ \mathit{Id} \mid \mathit{TApp}\ \mathit{Term}\ \mathit{Term}$$

The construct $\mathit{Var}\ i$ represents logical variable uniquely identified by i. All the Haskell values are *embedded* to the term data structure.

This approach has the merit of simplicity, but the resulting code is completely untyped.

Another possible approach is to define all the term types separately as what Claessen and Ljunglöf did in there typed embedding of Prolog into Haskell [2]. It is done in a way as follow[4]:

$$\begin{array}{l}\textbf{data}\ \mathit{TermB} = \mathit{Var}_1\ \mathit{Id} \mid \mathit{B}\ \mathit{Bool}\\ \textbf{data}\ \mathit{TermI}\ = \mathit{Var}_2\ \mathit{Id} \mid \mathit{I}\ \mathit{Int}\\ \vdots\end{array}$$

The terms here are completely typed. However, a drawback is that the large number of constructors are needed to represent logical variable, such as $\mathit{Var}_1\ i, \mathit{Var}_2\ i$ etc. This is definitely misleading since logical variables are supposed to be uninstantiated and should be able to instantiate to values of any type. This definition fails to achieve this because the logical variables are typed.

Our proposal is to have an embedded type term through the following:

$$\textbf{data}\ L\,\alpha\ =\ \mathit{Var}\ \mathit{Integer} \mid \mathit{Val}\,\alpha$$

This simple definition has several advantages. First of all, it is typed. A Haskell value is embedded to $L\ \alpha$ which is distinct from the embedded values of other types. In addition, the construct $\mathit{Var}\ i$ is truly a logical variable. It is uninstantiated and it is able to instantiate to any embedded Haskell values that is of type $L\,\alpha$. Moreover, it is extensible. An infinite number of Haskell types can be added to the system without any changes to the definition of the embedded term.

[4] The real implementation is far more complicated. We give a simple example here which reflects the design concept.

Definition 7. *A type τ is called* embedded *iff*

(1) $\tau = L\,\tau'$ where τ' is a basic type.

(2) $\tau = L\,\tau'$ where $\tau' = C\,\alpha_1\,\ldots\,\alpha_k$ where $\alpha_1 \ldots \alpha_k$ are embedded.

Example 2. $L\,Int$ and $L\,(C\,(L\,Int))$ are embedded while Int, $L\,(L\,Int)$, and $L\,(C\,Int)$ are not embedded.

Definition 8. *A type τ is called a* Haskell *type iff*

(1) τ is a basic type.

(2) $\tau = C\,\alpha_1\,\ldots\,\alpha_k$ where $\alpha_1 \ldots \alpha_k$ are Haskell types.

Definition 9. *Int and C Int are Haskell types while L Int, $L\,(L\,Int)$, and $L\,(C\,Int)$ are not Haskell types.*

Definition 10. *An embedding function $\mathcal{E} :: \tau \rightarrow \tau'$, where τ and τ' are the set of Haskell types and embedded types respectively, is defined as follow:*

If v is of an basic type α, then

$$\mathcal{E}\,v \;=\; Val\,v$$

If v is of a data type τ of the form

$$C\;v_1\;v_2\;\ldots\;v_n$$

then

$$\mathcal{E}(v) = Val\;(C\;\mathcal{E}(v_1)\;\mathcal{E}(v_2)\ldots\mathcal{E}(v_n))$$

Example 3.

$$\mathcal{E}(6) \;=\; Val\,6$$

$$\mathcal{E}(C\,5) \;=\; Val\,(C\,(Val\,5))$$

Projection function $\mathcal{P}$ which convert embedded values to Haskell values can be defined as the inverse of the embedding functions $\mathcal{P}$ with some exceptions. Before we go into the definition of projection, let us introduce the concept of groundness.

Definition 11. *An embedded value v is* ground *iff*

(1) $v = Val\ v$ where v is of a basic type.

(2) $v = Val\ v$ where $v = Val\ (C\ v_1\ v_2\ \ldots\ v_n)$ where $v_1\ v_2\ \ldots\ v_n$ are also ground.

In another words, an embedded values is ground iff it is not a $Var\ i$ and it does not have any $Var\ i$ as its component.

Example 4. $Val\ 4$ and $Val\ (C\ (Val\ 4))$ are ground while $Var\ 1$ and $Val\ (L\ (Var\ 1))$ are not ground.

Definition 12. *A projection function $\mathcal{P} :: \tau' \rightarrow \tau$, where τ and τ' are the set of Haskell types and corresponding embedded types respectively, is defined as follow:*

$\mathcal{P}(v)$=$\mathcal{E}^{-1}(v)$ *where v is ground*

$\mathcal{P}(v)$=*undefined where v is not ground.*

Example 5. $\mathcal{P}(\ Val\ 4) = 4$

$\mathcal{P}(\ Val\ (C\ (Val\ 7))) = (C\ 7)$

$\mathcal{P}(\ Val\ (C\ (Var\ 4)))$ = undefined

Definition 13. *A* term *has an embedded type.*

The definitions of embedding and projection provide essential foundation for the whole system. Since the whole CHR solver is built on top of terms, only with embedding and projection, an interface which connects Haskell and CHR can be created. It works by (a) embedding the Haskell data structure to term, (b) solving CHR constraints on term level, (c) projecting the result back to Haskell values.

Property 1. (*Idempotent*)

Project an embedded value returns the original value.

$$\mathcal{P} \circ \mathcal{E}(v)=v.$$

4.2 Substitution

After we have defined term embedding, the next step is to be able to define *substitution*. Usually, a substitution is defined as a list of tuples in which the

former elements in the tuple is substituted by the later ones. It has the type $[(var, term)]$. However, this does not work in our setting. Since terms in our system are typed, it is not possible to build up such a list because each list requires all the elements to be of the same type.

Example 6. Let us say we have a substitution $\mathcal{S}$=$\{x/1, y/$"*Winner*"$\}$.

Following the conventional way, we would attempt to represent $\mathcal{S}$ as

$$[((\mathit{Var}\ 1), (\mathit{Val}\ 1)), ((\mathit{Var}\ 2), (\mathit{Val}\ \text{"}\mathit{Winner}\text{"}))]$$

However, this does not work because *Val* 1 (of type *L Int*) does not match with

Val "Winner" (of type *L String*).

Our solution to this problem requires a type extension of Haskell, known as *Existentially quantified data constructors* (Jones, 1999) . We present this idea in the following sections.

Existentially quantified data constructors Consider the declaration:

data *Foo* = *forall* α . *MkFoo* α

The data type *Foo* has constructor *MkFoo*:

MkFoo :: *forall* α . $\alpha \rightarrow$ *Foo*

Notice that the type variable α in the type of *MkFoo* does not appear in the data type itself, which is plain *Foo*.Because of this, the following expression is fine:

$[\mathit{MkFoo}\ 5,\ \mathit{MkFoo}\ \text{`}c\text{'}] :: [\mathit{Foo}]$

On the same line of thinking,we introduced a new data type as follows:

data *Pair* = *forall* α . *Term* $\alpha \Rightarrow$ *MkPair Integer* α

In this definition, *Term* is a type class context imposed on α which basically says α can only take types from the *Term* type class.[5] The first parameter in this *Pair* of type *Integer* is used to uniquely identify a variable *Var i* which is to be substituted with the second parameter.

Definition of Substitution *Revisit Example 6*
With the data type *Pair*, we are able to define a new substitution list as follows:

$[\mathit{MkPair}\ 1\ (\mathit{Val}\ 1),\ \mathit{MkPair}\ 2\ (\mathit{Val}\ [\mathit{Val}\ 1,\ \mathit{Val}\ 2])] :: [\mathit{Pair}]$

[5] For more information of type class, please refer to Section [4.3.2] and (Jones, 1999).

Now we are ready to define substitution:

$$\textbf{type } \mathit{Subst} \ = \ [\mathit{Pair}]$$

The substitution $\mathcal{S}$=$\{x/1, y/[1,2]\}$ can be represented as:

$$\mathcal{S} \ = \ [\mathit{MkPair}\ 1\ (\mathit{Val}\ 1),\ \mathit{MkPair}\ 2\ (\mathit{Val}\ [\mathit{Val}\ 1,\ \mathit{Val}\ 2])]$$

4.3 Unification and Matching

Unification is another important operation which needs to be supported. We use it as the basic built-in solver of the CHR system. The unification algorithm is standard. The major issue here that it has to be overloaded to take argument of all the possible term types. We use a very distinctive feature of Haskell, namely *Type classes*, to resolve the problem. Before the implementation issues are brought up, let us first take a snapshot of polymorphism in Haskell.

Polymorphic Haskell Haskell incorporates polymorphic types (universally quantified in some way over all types). Polymorphic type expressions essentially describe families of types. For example, $(\forall\alpha)[\alpha]$ is a family of types consisting of, for every type α, the type of lists of α. Lists of integers (e.g.[1,2,3]), lists of characters ([‘a’,‘b’,‘c’]), even lists of lists of integers, etc, are all members of this family.

With the concept of polymorphic types, we can define functions like *count* which counts the number of elements in a list:

$$\begin{array}{ll} \mathit{length} & :: [\alpha] \rightarrow \mathit{Integer} \\ \mathit{length}\ [] & = 0 \\ \mathit{length}\ (x : xs) & = 1 \ + \ \mathit{length}\ xs \end{array}$$

This definition is almost self-explanatory. The kind of polymorphism presented here is called *parametric* polymorphism. It it more expressive than a monomorphically typed language. However, parametric polymorphism is only useful when types truly do not matter (for example, the *length* function really does not care what kind of elements are found in the list).

How about unification? Types do matter in this case. The reason is very simple: unification needs to deconstruct its arguments and try to unify their sub-components. This process is apparently type dependent. Therefore, we need an overloaded function which behave differently for each type. This kind of polymorphism is called *ad hoc* polymorphism, or *overloading*. One special feature of Haskell, *Type Classes*, provides a structured way to control it.

Type Classes and Overloading Haskell type classes are roughly similar to a Java interface. Like an interface declaration, a Haskell type class declaration defines a protocol for using an object rather than defining an object itself. Type classes allow us to declare which types are *instances* of which class, and to provide definitions of the overloaded *operations* associated with a class. For example, let us define a type class containing an equality operator:

class *Eq* α **where**
$(==) :: \alpha \to \alpha \to Bool$

Here *Eq* is the name of the class being defined, and == is the single operation in the class. This declaration may be read "a type α is an instance of the class *Eq* if there is an (overloaded) operation ==, of the appropriate type, defined on it."This kind of type classes are also called *parametric* type classes because it takes type variables, in this case α, as parameters.

So far so good. But how do we specify which types are instances of the class *Eq*, and the actual behavior of == on each of those types? This is done with an *instance* declaration. For example:

instance *Eq Integer* **where**
$x == y = x \; `integerEq' \; y$

The definition of == is called a *method*. The function *integerEq* happens to be the primitive function that compares integers for equality, but in general any valid expression is allowed on the right-hand side. The overall declaration is essentially saying:"The type *Integer* is an instance of the class *Eq*, and here is the definition of the method corresponding to the operation ==."

Recursive types such as *List* can also be handled:

instance $(Eq\ \alpha) \Rightarrow Eq\ List\ \alpha$ **where**
$Nil == Nil = True$
$(Cons\ x\ xs == Cons\ x'\ xs') = (x == x')\ \&\&\ (xs == xs')$
$\_ == \_ = False$

In the code above, "_" is a wildcard which is able to match any pattern.[6]

The $\Rightarrow$ imposes a constraint that a type α must be an instance of the class *Eq*. It is written *Eq* α. Thus *Eq* α is not a type expression, but rather it expresses a constraint on a type, and is called a *context*. The context is necessary in the above example because the elements in the list (of type α) are compared for equality in the second line. This additional constraint is essentially saying that we can compare lists of α's for equality as long as we know how to compare α's for equality.

Unification with Type Classes In a very similar way to how == is defined, we define *unify*. Firstly, we introduce a type class named Term which contains all the term types.

class *Term* α **where**
unify :: $Subst \to \alpha \to \alpha \to Subst$

There is one operation *unify* defined in this class. It takes in a substitution and two terms of the same type, then returns an updated substitution or an error if the unification fails. I will firstly illustrate how *unify* can be defined by some examples before giving out the generic algorithm for generating *unify*.

instance *Term* $(L\ Int)$ **where**

[6] For more details on patten matching,please refer to (Jones, 1999).

```
unify subst (Val x) (Val y)
  | x == y = return subst
  | otherwise = errorE "fail : No mgu" idSubst
unify subst (Var x) (Val y) = return (updateSubst [(Pair x (Val y))] subst)
unify subst (Val y) (Var x) = unify subst (Var x) (Val y)
unify subst (Var x) (Var y) =
  if x == y then return subst
            else return (updateSubst [(Pair x (Var y))] subst)
```

This is the *unify* method for *L Int*. The function *updateSubst* will apply its first argument which is a substitution onto its second argument which is another substitution. Next example is the *unify* method of *L Employee*.

```
instance Term (L Employee) where
  unify subst (Val (Academic s i)) (Val (Academic s' i')) = do sub ← unify subst s s'
                                                               return unify sub i i'
  unify subst (Val (Nonacademic s)) (Val (Nonacademic s')) = return unify subst s s'
  unify subst (Var x) (Val y) = return (updateSubst [(Pair x (Val y))] subst)
  unify subst (Val y) (Var x) = return (unify subst (Var x) (Val y))
  unify subst (Var x) (Var y) =
    if x == y then return subst
              else return (updateSubst [(Pair x (Var y))] subst)
  unify subst _ _ = errorE "fail : No mgu" idSubst
```

In this definition, the data type is deconstructed to allow the sub-components are unified in a sequence. The result of the unification of the first pair of sub-components are applied to the unification of the second pair and so on. The last line states that except for the conditions listed above, all the others lead to failure of unification. The reader may notice that we do not perform *occur check* here. This is because the type checking in the previous phase has ensured that there is no such possibility for a variable to be unified with a term that contains it.

With overloading, *unify* can be easily defined for each terms. However, as our system allows the programmer to embed any user defined Haskell data type as a term, it is not tedious to ask the user to write the *unify* function for each term they introduced. A better solution here is automatic code generation. We notice from the *unify* for *L Employee* and *L Int*, only the first two clauses of the definition is unique. The rest are standard for all the data types. We present a generic algorithm for the generation of *unify*:

Algorithm
Consider the *unify* generation function $\mathcal{G}_u : D \rightarrow U$ where D is an embedded term and U is the *unify* function defined on it.
Let D be of the form $L\ C$
where

$$\textbf{data}\ C = C_1\ \tau_{11}\ \tau_{12}\ \ldots\ \tau_{1i} \mid \ldots \mid C_k\ \tau_{k1}\ \tau_{k2}\ \ldots\ \tau_{kj}$$

then
$\mathcal{G}_u(D) =$

```
instance Term (L C) where
  unify subst (Val (C_1 x_1 x_2 ... x_i)) (Val (C_1 x'_1 x'_2 ... x'_i)) =
      do s_1 ← unify subst x_1 x'_1
         s_2 ← unify s_1 x_2 x'_2
         ⋮
         return unify s_{i-1} x_i x'_i
  ⋮
  unify subst (Val (C_k x_1 x_2 ... x_j)) (Val (C_k x'_1 x'_2 ... x'_j)) =
      do s_1 ← unify subst x_1 x'_1
         s_2 ← unify s_1 x_2 x'_2
         ⋮
         return unify s_{j-1} x_j x'_j
  unify subst (Var x) (Val y) = return (updateSubst [(Pair x (Val y))] subst)
  unify subst (Val y) (Var x) = (unify subst (Var x) (Val y))
  unify subst (Var x) (Var y) =
    if x == y then return subst
              else return (updateSubst [(Pair x (Var y))] subst)
  unify subst _ _ = errorE "fail : No mgu" idSubst
```

4.4 Matching

Matching is another essential operation for CHR system. It is used to check the matching of heads of CHR rules against the constraints in the goal.

Definition 14. *Let t_1, t_2 be two terms.*

We say t_1 matches *t_2 iff $t_1 \doteq \phi t_2$ for some substitution ϕ ($\doteq$ stands for syntactic equality).*

Matching is essentially very similar to unification. It is sometimes called *one-way* unification because only one side is allowed to be substituted. In our case, the terms in goal are not allowed to be substituted[7].

In a real implementation, we reuse the function unify for the definition of matching by replacing all variables in the terms in the goal by a construct that behave exactly like a value. As the reader may expect, we add matching as an operation of class term.

4.5 Type Class, again

We have mentioned so much about the implementation of unification using type classes. But how about the embedding and projection of the terms? Intuitively, we would try to define it as an operation in the term class.

[7] For more details on semantics of CHR, please refer to [1].

class *Term* α **where**
  unify :: *Subst*$\rightarrow \alpha \rightarrow \alpha \rightarrow$ *Subst*
  match :: *Subst*$\rightarrow \alpha \rightarrow \alpha \rightarrow$ *Subst*
  embed :: $L\ \alpha \rightarrow \alpha$
  proj :: $\alpha \rightarrow L\ \alpha$

This approach works well for basic type such as *Int*.

instance *Term Int* **where**
  embed i = *Val* i
  proj (*Val* i) = i

Then how about [*Int*]? What should be the parameter of such an instance? *Int*, [*Int*], and [*L Int*] all do not work. The reason here is the input/output type relationships of *embed* and *proj* are dependent on the type. To relate them, we shall attempt to make use of *multi-parametric* type classes:

class *Term* $\alpha\ \beta$ **where**
  embed :: $\alpha \rightarrow \beta$
  proj :: $\beta \rightarrow \alpha$

In addition, we declare instances of the class. For example:

instance *Term Int* (*L Int*) **where**
  embed i = (*Val* i)
  proj (*Val* i) = i

This seems to be fine. But does it work? We try to execute (*embed* 0). The interpreter yields the following error:

```
<interactive>:1:

    No instance for (Term Int b)

    arising from use of ‘embed’ at <interactive>:1

    In the definition of ‘it’: (embed 0)
```

This is because when we call *embed* with the argument 0, only type parameter α in the class term is instantiated to *Int*. The interpreter has no clue of what β is. As a result, it will try to find a instance which have the signature (*Term Int* β) which of course does not exist.

To resolve this, we need to make use another Haskell type extension, namely *functional dependencies* [9]. It is used to declare explicit functional dependencies between the parameters of a predicate. With it, we can declare a class as:

class *Term* $\beta \Rightarrow$ *Embed* $\alpha\ \beta \mid \alpha \rightsquigarrow \beta$ **where**
  embed :: $\alpha \rightarrow \beta$
  proj :: $\beta \rightarrow \alpha$

Then we can define instances of it using:

instance *Embed Int* (*L Int*) **where**
  embed i = (*Val* i)
  proj (*Val* i) = i

With the functional dependency explicitly declared in the class definition, we can now process (*embed* 0). The compiler will be able to associate β with the output type of (*embed Int*) which is *L Int*. This matches precisely with the signature of the instance declared. Similarly, instance of [*Int*] can be defined as:

```
instance Embed Int (L [L Int]) where
  embed is = (Val (map embed is))
  proj (Val is) = map proj is
```

5 Putting Everything Together

After the foundation is laid, we can now describe the implementation for the CHR solver. In this chapter, I will discuss the implementation issues and interaction between the CHR solver and the host language.

5.1 Solver with Monadic Programming

The implementation of the solver follows closely the operational semantics of CHR. It is essentially a transition system which constantly checks every CHR against a global store. If there is a rule whose head matches a set of user-defined constraints in the store,and some other conditions, like guard etc, are satisfied, the store will be updated. If there is no applicable rules or a failure of the built-in store, the solver terminates.

This state transition process can be modelled perfectly by using monadic programming in Haskell. *Monads* arose in category theory. It could be used to model a wide variety of language features, including non-termination, state, exception[8], continuations, and interaction. In our implementation, we make heavy use of state monad to model the global store of the solver. In this section, we will focus on the system design instead of monad itself. Readers interested in knowing more about monad, please refer to [9] for a formal description of the topic.

From now on, a reader should imagine there is an implicit global store which all the functions defined within its scope are able to access and update. A monadic function can be seen as an action. It has some effects and also returns a value. In our case, the effect is the possible updating of the global store, the return value is dependent on individual functions. The following set of actions(functions) are defined in our system. Only signatures are shown here:

$$
\begin{aligned}
&\mathit{initialize} :: \mathit{Goal} \rightarrow M\ () \\
&\mathit{add} :: \mathit{Goal} \rightarrow M\ () \\
&\mathit{introduce} :: M\ () \\
&\mathit{check} :: M\ \mathit{Bool} \\
&\mathit{remove} :: [\mathit{Index}] \rightarrow M\ \mathit{Bool} \\
&\mathit{findmatchS} :: [\mathit{UCons}] \rightarrow [\mathit{BCons}] \rightarrow M\ ([\mathit{BCons}], [\mathit{Index}]) \\
&\mathit{findmatchP} :: [\mathit{UCons}] \rightarrow [\mathit{BCons}] \rightarrow M\ ([\mathit{BCons}], [\mathit{Index}])
\end{aligned}
$$

In the above signatures, M is a state monad. The type following it is the type of the return value. () is seen as void type. *initialize* initializes the global store with a goal. *add* adds the body of an applied rule to the goal store. Since the body is a mixture of user-defined and built-in constraints, it can also be seen as a goal. *introduce* transports all the user-defined and built-in constraints in the goal store to the user-defined and built-in store respectively. This action can be seen as a closure of the combined effect of the introduce and solve step in the operational semantics. *check* checks the satisfaction of the built-in store

[8] The error message in the function *unify* discussed in the previous chapter is handled by an exception monad.

and return the result as a *Bool* value. *remove* removes a set of user-defined constraints indexed by their names from the store due to an application of a simplification rule. *findmatch* takes the head and guard of a rule which is a set of user-defined and built-in constraints respectively. It will exhaustively search the built-in constraint store with backtrack for a matching of the head with the guard satisfied. If it succeeds, a set of equality constraints follows from the matching and indexes of the matched user-defined constraints are returned. The commutativity of constraints in the rule head is handled by permutation. *findmatchP* is the *findmatch* variant for propagation rules. This differs from *findmatchS* by additional checking on the token store for non-termination.

With the basic actions listed above, we can define a rule as an action. A simplification rule can be defined as:

```
simp :: Rule → M Bool
simp r = do (R head guard goal) ← rename r
            res ← findmatchS head guard
            case res of (idSubst, []) → M(λ s → return (s, False))
                        (eqs, indexes) → do { remove indexes
                                            ; add (combine eqs goal)
                                            ; introduce
                                            ; check}
```

In the above Haskell function *do* notation is used to construct a sequence of actions. ← assigns the return value of the action on the right hand side to the variable on the left hand side. *simp* takes in a set of rule specifications include the head, guard, and body of the rule and constructs a simplification rule. The simplification rule is a sequence of actions. It first renames the variables in the rule followed by trying to find a matching for the rule head. If the finding matching is not successful, nothing will be done to the global store. *False* will be returned as an indication of this failure. If a successful matching is found, the matched constraints in the user-defined constraint store are removed. Next, the body of the rule together the equality constraints from the matching are added to the goal store. After that, they are introduced and the built-in constraint store is checked. A propagation rule can be constructed in a very similar way to a simplification rule with *findmatchP* in the place of *findmatchS* and no removal of constraints.

After all the rules are constructed, they will be gathered as a list of rules. The solver will initialize the store and invoke a loop to attempt all the rules one by one on the store until no rule is applicable. This is signaled when all the rules return *False*.

5.2 Interaction with the Host language

By now, we have a CHR solver embedded in Haskell. In this section, we will discuss how the CHR solver interacts with the host language.

The ground has been prepared as we have functions *embed* and *project* which are able to do the conversion between Haskell values and embedded terms. What left is to define a query which queries the store after the CHR solving has completed its task. There are two queries currently defined. The signatures are:

```
query :: L α → M (Maybe L α)
queryAll :: L α → M String
```

query takes in a variable as argument and query the value of the variable in a final store. The return type is a *Maybe* type. Maybe is a type in Haskell which can be either *Nothing* or a value. If the final store failed or the queried variable are not instantiated, *query* returns *Nothing*. Otherwise, the instantiated value is returned. *queryAll* will return the simplified built-in store as a string for display. The local variables are projected out for clarity.

With the query functions, we can call the CHR solver form Haskell functions by supplying a goal and a set of CHRs. After the solving has been done, the programmer can query the variable of interest and make use of the value returned for further computation. The following fragment of code gives us a taste of how this can be done:

```
run = runM (do
                ⋮
                solve aGoal
                if success
                then
                    do r ← (query "x")
                       case r of
                              Nothing → continueSolve moreGoal chrs
                              Just x → f (proj x))
                else
                    print "Goal failed"
```

In this piece of code, the programmer programs in Haskell and calls the CHR solver within the Haskell function. The programmer passes a goal named *aGoal* to the CHR solver. If the solving is successful, the value of x is queried. Note that x should be a variable in the initial goal *aGoal*, otherwise, this solving and query do not make sense. After the value of x is returned, the program checks whether it has been initialized during the CHR solving. If so, the value is projected and passed to another Haskell function. Otherwise, more goals are passed to the solver in order to produce a more determined result. *continueSolve* behaves exactly like *solve* except no store initialization is done. This example demonstrate an important feature of the CHR solver namely *incremental behaviour*. Though the CHR language definition itself is incremental, it is very hard to utilize this feature if it is implemented as a stand alone solver. In our system, as seen from the above example, the incremental behaviour is achieved in a straight forward way. Another point to be emphasize is the elegance of the syntax. This is another advantage in our language which comes from the approach of embedding in a strongly typed functional programming language.

6 Applications

6.1 Boolean Solver

The domain of Boolean constraints includes the constants 0 for falsity, 1 for truth and the usual logical connectives of propositional logic, e.g. And, Or, Neq, Imp, $Exor$, modelled here as relations. The behaviour of an and-gate can be defined with HCHRs:

$$\begin{array}{l}
\textbf{data}\ Pred\ =\ And\,(Int, Int, Int)\\
And\,(x,\ y,\ z)\ \Leftrightarrow\ x\ =\ 0\ \mid\ z\ =\ 0;\\
And\,(x,\ y,\ z)\ \Leftrightarrow\ y\ =\ 0\ \mid\ z\ =\ 0;\\
And\,(x,\ y,\ z)\ \Leftrightarrow\ x\ =\ 1\ \mid\ y\ =\ z;\\
And\,(x,\ y,\ z)\ \Leftrightarrow\ y\ =\ 1\ \mid\ x\ =\ z;\\
And\,(x,\ y,\ z)\ \Leftrightarrow\ z\ =\ 1\ \mid\ x\ =\ 1,\ y\ =1;\\
And\,(x,\ y,\ z)\ \Leftrightarrow\ x\ =\ y\ \mid\ y\ =\ z;
\end{array}$$

For example, the first rule stats that when it is known that the first input argument of constraint $And\,(x,\ y,\ z)$ is 0, it can be reduced by asserting that output z must be 0, too.

Example 7. Consider the predicate Add taken from the well-know full-adder circuit. It adds three single digits binary numbers to produce a single number consisting of two digits:

$$\begin{array}{l}
Add\,(i_1,\ i_2,\ i_3,\ o_1,\ o_2)\ =\\
\qquad Xor\,(i_1,\ i_2,\ x_1),\ And\,(i_1,\ i_2,\ a_1),\\
\qquad Xor\,(x_1,\ i_3,\ o_2),\ And\,(i_3,\ x_1,\ a_2),\\
\qquad Or\,(a_1,\ a_2,\ o_1).
\end{array}$$

The query $Add\,(i_1,\ i_2,\ 0,\ 0,\ o_2)$ will reduce to $i_1 = 1, i_2 = 1, o_2 = 0$.

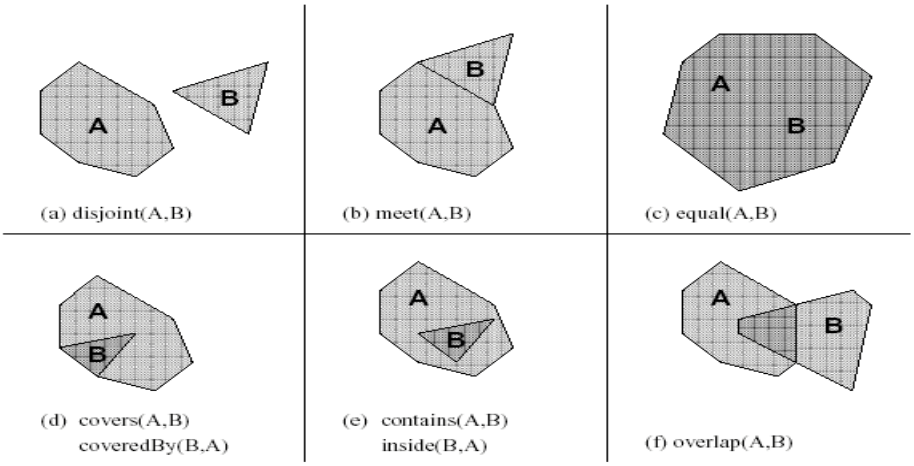
Another query $Add\,(i_1,\ i_2,\ 0,\ o_1,\ 1)$ gives $i_1 = 1, i_2 = 1, o_1 = 0$

6.2 Qualitative Spatial Reasoning

Consider now the following problem.

> Two houses are connected by a road. The first house is adjacent to its garden's boundary while the second house is surrounded by its garden. The question is what can we conclude about the relation between the second garden and the road.

To analyse such problems we need to consider relative positions between contiguous objects. This brings us to the eight possibilities that are summarized in the figure below. These eight relations are usually called *spatial relations*.



Eight spatial relations[9]

We can model this problem in HCHR, as follows:

data *Object* = *House Host* | *Road* | *Garden*
data *Person* = *Person Name*
type *Name* = *String*
type *Rcc* = *Disjoint* | *Meet* | *Equal* | *Covers* | *CoveredBy* | *Contains* | *Inside* | *Overlap*
data *Pred* = *C* (*Object*, *Object*, *Rcc*)
H_1 = *House John*
H_2 = *House Mike*
R = *Road*
G_1 = *Garden*
G_2 = *Garden*

[9] (d) and (c) in this figure represents two different spatial relations each.

The CHRs can be defined. Example of rules are:

$$C\,(o_1, o_2, x)\,,\ C\,(o_2, o_3, x),\ C\,(o_1, o_3, \mathit{Equal}) \implies x \neq \mathit{Inside},\ x \neq \mathit{coveredBy},\ x \neq \mathit{contains}, x \neq \mathit{covers};$$
$$C\,(o_1, o_2, x)\,,\ C\,(o_2, o_3, \mathit{Equal}),\ C\,(o_1, o_3, z) \implies x = z;$$
$$C\,(o_1, o_2, \mathit{Contains})\,,\ C\,(o_2, o_3, y),\ C(o_1, o_3, \mathit{Disjoint}) \implies y = \mathit{Disjoint};$$

The goal for the problem at the beginning of the section is defined as:

$$C\,(H_1,\ G_1,\ \mathit{CoverBy});\ C\,(H_2,\ G_2,\ \mathit{Inside});$$
$$C\,(H_1,\ H_2,\ \mathit{Disjoint});\ C\,(H_1,\ R,\ \mathit{Meet});$$
$$C\,(H_2,\ R,\ \mathit{Meet});\ C\,(G_2,\ R,\ x);$$

Then the solver will be able to reduce the set of values x can take to $\{\mathit{Covers},\ \mathit{Overlap}\}$.

This example justifies the design goal of HCHR. It is not meant to be a constraint programming language. Instead, it is seen as an extension of Haskell. Thus, it is done in a way that:

- The syntax should be as similar as Haskell as possible.
- Haskell types and values should be able to referred by the CHR solver as it is.
- The result of the CHR solving should be directly accessible from Haskell.

We can see from the above example, the HCHR code is programmed in a style which is very similar to Haskell. The Haskell value, H_1 for instance, is passed into the goal directly. The power of the CHR solving can be used by Haskell program in a straight forward way. Moreover, the rules are type checked according the data types declared in Haskell. All these features make HCHR a very powerful and convenient tool for Haskell programmers.

7 Conclusion and Future Work

In this thesis, the design and implementation issues of HCHR have been presented. We started with the syntax and semantics of CHR, followed by type checking of CHR declarations. We developed a simple type system. To the best of our knowledge, this has not been achieved before. We proposed a novel way of embedding which works very well. This terminology can be widely applied in the area of generic programming. We have explored a wide range of Haskell language features, including existentially quantification and type classes, to make the embedding successful. We showed our implementation of CHR in monadic framework. The interaction between Haskell and CHR is discussed, followed by some possible applications. We have shown that in HCHR, several advantages of CHR, such as incremental behaviour, can be fully utilized.

There are also some enhancements of HCHR we are currently looking into. One of them is to be able to call Haskell functions from CHR rules . This has been shown to be important in many applications. However, this is non-trivial. If function calls instead of terms are allowed in the rules, we will not be able to build the unifiers because unification is defined only on terms. One proposal to avoid it is to have specific matching function for each rule instead of the generic approach used in the current implementation. Besides this, we still need to consider some delayed application techniques which may upset the CHR semantics.

We may need to consider some external solvers other than the HERBRAND solver that we are presently using. The idea is to use overloading to support several independent low level solvers on different domains to coexist in the system so a wilder range of problems can be handled.

Another possible direction is to extend the typed CHR to a polymorphic type system, like Haskell. This may be useful to allow CHR programs to scale up. In this case, we need to provide soundness proof as it is no long trivial as in the current case.

Last but not least, we will be looking for more application of the HCHR language. One particular area of interest is program analysis. As many researcher use functional languages to express program analysis, and many of the analysis require solution in constraint form, we believe HCHR has a bright future.

References

1. S. Abdennadher. Operational semantics and confluence of constraint propagation rules. In *Proc. of CP'97*. Springer LNCS, 1997.
2. K. Claessen. and P. Ljunglöf. Typed logical variables in haskell. In *Proc. of Haskell '02*. ACM Press, 2000.
3. T. Frühwirth. Theory and practice of constraint handling rules. *Special Issue on Constraint Logic Programming (P. Stuckey and K. Marriot, Eds.), Journal of Logic Programming*, 37(1-3):95–138, October 1998.
4. T. Frühwirth and P. Brisset. High-level implementations of constraint handling rules. Technical Report ECRC-95-20, ECRC Munich, 1995.
5. M. Hanus. Curry: An integrated functional logic language, 2002.
6. R. Hinze. Prolog's control constructs in a functional setting - axioms and implementation. *International Journal of Foundations of Computer Science*, 12(2):125–170, 2001.
7. P. Hudak. Modular domain specific language and tools. In *Fifth International Conference on Software Reuse,1998*, 1998.
8. J. Jaffar and M. J. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19(20):503–581, 1994.
9. S. P. Jones and J. Hughs. www.haskell.org.
10. J. W. Lloyd. Declarative programming in escher. Technical Report CSTR-95-013, University of Bristol, 1995.
11. R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–275, 1978.
12. J. Moreno-Navarro and M. Roderiguez-Artalejo. Logic programming with functions and predicates: The language babel. *Journal of Logic Programming*, 12(3):191–223, 1992.
13. M. Saft S. Abdennadher, E. Krämer and M. Schmauss. Jack: A java constraint kit. *Electronic Notes in Theoretical Computer Science*, 64, 2002.
14. M. Sulzmann. www.comp.nus.edu.sg/ sulzmann/chr.