Pros and Cons of Using CHR for Type Inference

János Csorba, Zsolt Zombori, and Péter Szeredi

Department of Computer Science and Information Theory
Budapest University of Technology and Economics
{csorba,zombori,szeredi}@cs.bme.hu

Abstract. We report on using logic programming and in particular the Constraint Handling Rules extension of Prolog to provide static type analysis for the Q functional language. We discuss some of the merits and difficulties of CHR that we came across during implementation of a type inference tool.

Keywords: logic programming, static type inference, CSP, CHR, Q language

1 Introduction

Our paper presents an application of Constraint Handling Rules (CHR) for the type analysis of the Q functional language. We implemented the `qtchk` *type inference* tool, which has been developed in a collaborative project between Budapest University of Technology and Economics and Morgan Stanley Business and Technology Centre, Budapest.

The main goal of the type inference tool is to detect type errors and provide detailed error messages explaining the inconsistency. The `qtchk` program infers the possible types of all expressions in the program. Consequently, for any syntactically correct Q program the analyser will detect type inconsistencies, and will assign a type to each type-consistent expression of the Q program at hand.

We reported on the main issues of the type inference application in our ICLP paper [8]. There we described type inference as a Constraint Satisfaction Problem (CSP) and presented how the task of type analysis can be mapped onto a CSP. In the present paper we focus on the implementation details: how we solved this problem using the Constraint Handling Rules extension of Prolog [4, 5].

In Section 2 we briefly introduce the Q language and provide some background information. In Section 3 we give an overview of the main issues and of the implementation details of the constraint based type inference tool for Q. Section 4 is devoted to the discussion of some difficulties in using CHR. In Section 5 we provide an evaluation of using CHR.

2 Preliminaries

In this section we give some background to our work following [8], where the reader can find more details. We first introduce the Q programming language, and then give an overview of the type language developed for Q. Finally we discuss the mapping of a type inference task into a Constraint Satisfaction Problem.

2.1 The Q Programming Language

Q is a highly efficient vector processing functional language, which is well suited to performing complex calculations quickly on large volumes of data.

Types Q is a strongly typed, dynamically checked language. This means that while each variable, at any point of time, is associated with a well defined type, the type of a variable is not declared explicitly, but stored along its value during execution. The most important types are as follows:

– **Atomic types** in Q correspond to those in SQL with some additional date and time related types that facilitate time series calculations.[1]
– **Lists** are built from Q expressions of arbitrary types, e.g. `(1;2.2;‘abc)` is a list comprising two numbers and a symbol.
– **Dictionaries** are a generalisation of lists and provide the foundation for tables. A dictionary is a mapping that is given by exhaustively enumerating all domain-range pairs.
– **Tables** are lists of special dictionaries that correspond to SQL records.
– **Functions** correspond to mathematical mappings specified by an algorithm.

Main Language Constructs Q being a functional language, functions form the basis of the language. A function is composed of an optional parameter list and a body comprising a sequence of expressions to be evaluated. Function application is the process of evaluating the sequence of expressions obtained after substituting actual arguments for formal parameters. For example, the expression `f: {[x] $[x>0;sqrt x;0]}` defines a function of a single argument x, returning $\sqrt{x}$, if $x > 0$, and 0 otherwise.

Some built-in functions (dominantly mathematical functions) with one or two arguments have a special behaviour called *item-wise extension*. Normally, the built-in functions take atomic arguments and return an atomic result of some numerical calculation. However, these functions extend to list arguments item-wise. If a unary function is given a list argument, the result is the list of results obtained by applying the function to each element of the input list. A binary function with an atom and a list argument evaluates the atom with each list element. When both arguments are lists, the function operates pair-wise on elements in corresponding positions. Item-wise extension applies recursively in case of deeper lists, e.g. `((1;2); (3;4)) + (0.1; 0.2) = ((1.1;2.1); (3.2;4.2))`

While being a functional language, Q also has imperative features, such as multiple assignment[2] of variables and loops.

2.2 Type Language for Q

In this subsection we describe the type language developed for Q. We allow polymorphic type expressions, i.e., any part of a complex type expression can be

[1] Q has the following 16 atomic types: `boolean`, `byte`, `short`, `int`, `long`, `real`, `float`, `char`, `symbol`, `date`, `datetime`, `minute`, `second`, `time`, `timespan`, `timestamp`.
[2] Assignment is denoted by a colon, e.g. `x:x*2` doubles the value of the variable `x`.

replaced with a variable. Expressions are built from atomic types and variables using type constructors. The abstract syntax of the type language – which is at the same time the Prolog representation of types – is as follows:

```
TypeExpr =
     AtomicTypes     | TypeVar     | symbol(Name)    | any
   | list(TypeExpr) | tuple([TypeExpr,...,TypeExpr])
   | dict(TypeExpr, TypeExpr)    | func(TypeExpr, TypeExpr)
```

AtomicTypes This is shorthand for the 16 atomic types of Q. Furthermore, the `numeric` keyword is used to denote a type consisting of all numeric values.

TypeVar represents an arbitrary type expression, with the restriction that the same variables stand for the same type expression. Type variables make it possible to define polymorphic type expressions, such as `list(A) -> A` (a function mapping a list of a certain type to a value of the given type) and `tuple([A,A,B])`.

`symbol(`*Name*`)` The named symbol type is a degenerate type, as it has a single instance only, namely the provided symbol. Nevertheless, it is important because in order to support certain table operations, the type reasoner needs to know what exactly the involved symbols are.

`any` This is a generic type description, which denotes all data structures allowed by the Q language.

`list(`*TE*`)` The set of all lists with elements from the set represented by *TE*.

`tuple([`TE_1`, ..., `TE_k`])` The set of all lists of length k, such that the i^{th} element is from the set represented by TE_i.

`dict(`TE_1`,`TE_2`)` The set of all dictionaries, defined by an explicit association between domain list (TE_1) and range list(TE_2) via positional correspondence.

`func(`TE_1`, `TE_2`)` The set of all functions, such that the domain and range are from the sets represented by TE_1 and TE_2, respectively.[3]

2.3 Type Inference as a Constraint Satisfaction Problem

In this subsection we give an overview of our approach of transforming the problem of type reasoning onto a CSP. Type reasoning starts from program code that can be seen as a complex expression built from simpler expressions. Our aim is to assign a type to each expression appearing in the program in a coherent manner. The types of some expressions are known immediately, while other kinds of information are provided by the program syntax, which imposes restrictions between the types of certain expressions. The aim of the reasoner is to assign a type to each expression that satisfies all the restrictions.

We associate a CSP variable with each sub-expression of the program. Each variable has a domain, which initially is the set of all possible types. Different type restrictions can be interpreted as constraints that restrict the domains of some variables. In this terminology, the task of the reasoner is to assign a value to each variable from the associated domain that satisfies all the constraints.

[3] To help readability, we often use the notation `A -> B` instead of `func(A,B)`.

However, our task is more difficult than a classical CSP, because there are infinitely many types (e.g. tuples can be of arbitrary length), which cannot be represented explicitly in a list. Representing infinite domains is a challenge for performing type reasoning.

Partial Ordering We say that type expression T_1 is a subtype of type expression T_2 ($T_1 \leq T_2$) if, and only if, all values that belong to T_1 also belong to T_2. The subtype relation determines a partial ordering over the type expressions.

For example, consider the `tuple([int,int])` type which represents all lists of length two, where both elements are integers. It is obvious that every value that belongs `tuple([int,int])` also belongs to `list(int)`, i.e., the type expression `tuple([int,int])` is a subtype of `list(int)`.

It is very easy to check whether the subtype relation holds between two type expressions. For atomic type expressions this is immediate. Complex type expressions can be checked using some simple recursive rules. For example, `list(A)` is a subtype of `list(B)` if, and only if, `A` is subtype of `B`.

Finite Representation of the Domain The domain of a variable is initially the set of all the types, which can be constrained with different upper and lower bounds, based on the partial ordering.

An upper bound restriction for variable X_i is a list $L_i = [T_{i1}, \ldots, T_{in_i}]$, meaning that the upper bound of X_i is $\bigcup_{j=1}^{n_i} T_{ij}$, i.e., the type of X_i is a subtype of some element of L_i. Disjunctive upper bounds are very common and natural in Q, for example, the type of an expression might have to be either `list` or `dict`. The conjunction of upper bounds is easily described by having multiple upper bounds. If variable X_u gets a new upper bound L_v (e.g. because it turns out that variable X_v is a subtype of X_u, and so X_u inherits the upper bound of X_v), this means that the value of X_u has to be in $\bigcup(T_{uj} \bigcap T_{vk})$, for all $1 \leq j \leq n_u$ and $1 \leq k \leq n_v$.

A lower bound restriction for variable X_i is a single type expression T_i, meaning that T_i is a subtype of X_i. For lower bounds, it is their union which is naturally represented by having multiple constraints: if X has two lower bounds T_1 and T_2, then $T_1 \cup T_2$ is a subtype of X. We do not use lists for lower bounds, so we cannot represent the intersection of lower bounds. We chose this representation because no language construct in Q yields a conjunctive lower bound.

3 Implementing Type Inference using CHR

We built a Prolog program called `qtchk` that implements the type reasoning as a Constraint Satisfaction Problem using the Constraint Handling Rules library. It runs both in SICStus Prolog 4.1 [6] and SWI Prolog 5.10.5 [7]. It consists of over 8000 lines of code[4]. Q has many irregularities and lots of built-in functions (over 160), due to which a complex system of constraints had to be implemented

[4] We are happy to share the code over e-mail with anyone interested in it.

using over 60 constraints. The detailed user manual for `qtchk` can be found in [3] that contains lots of examples along with the concrete syntax of the Q language.

The system has two main components: a parser and a type inference engine. The parser builds an abstract tree (AST) representation of the code, where each node represents a sub-expression. Afterwards, we traverse the AST and formulate CHR constraints on which type inference is performed. Both phases detect and store errors, which are presented to the user. In this section we focus on the implementation of the CHR based type inference component.

3.1 Representing variables

All subexpressions of the program are associated with CSP variables. If some constraint fails, we need to know which expression is erroneous in order to generate a useful error message. If the arguments of the constraints are Prolog variables, we do not have this information at hand. Hence, instead of variables we use identifiers `ID = id(N,Type,Error)`[5] which are Prolog terms with three arguments: an integer `N` which uniquely identifies the corresponding expression, the type proper `Type` (which is a Prolog variable before the type is known) and an error flag `Error` which is used for error propagation. We use the same representation for type variables in polymorphic types, e.g. the type `list(X)` may be represented by `list(id(2))`.

3.2 Constraint Reasoning

After parsing, the type analyser traverses the abstract syntax tree and imposes constraints on the types of the subexpressions of the program. The constraints describing the domain of a variable are particularly important, we call them *primary constraints.* These are the upper and lower bound constraints. We will refer to the rest of the constraints as *secondary constraints.* Secondary constraints eventually restrict domains by generating primary constraints, when their arguments are sufficiently instantiated (i.e., domains are sufficiently narrow).

Our aim is to eventually eliminate all secondary constraints. If we manage to do this, the domains described by the primary constraints constitute the set of possible type assignments to each expression. In case some domain is the empty set, we have a type error. Otherwise, the program is considered type correct.

If the upper and lower bounds on a variable determine a singleton set[6], then we know the type of the variable and we say that it is *instantiated.* If all arguments of a secondary constraint are instantiated, then there are two possibilities. If the instantiation satisfies the constraint, then the latter can be removed from the store. Otherwise, the constraint fails.

[5] In order to make the examples easier to read, we will omit the two variable arguments of the `id/3` compound term, i.e. use `id(N)` instead of `id(N,Type,Error)`.
Also note that we will use the terms "variable" and "identifier" interchangeably.

[6] This is the case, e.g., when the lower and upper bounds are the same, or when there is an atomic upper bound.

Constraint reasoning is performed using the Constraint Handling Rules library of Prolog. In the next two paragraphs we describe how these constraints interact with each other.

Interaction of Primary Constraints Primary constraints represent variable domains. If a domain turns out to be empty, this indicates a type error and we expect the reasoner to detect this. Hence, it is very important for the constraint system to handle primary constraints as "cleverly" as possible. For this, we formulated rules to describe the following interactions on primary constraints:

– Two upper bounds on a variable should be replaced with their intersection.
– Two lower bounds on a variable should be replaced with their union.
– If a variable has an upper and a lower bound such that there is no type that satisfies both, then the clash should be made explicit by setting the upper bound to the empty set.
– Upper and lower bounds can be polymorphic, i.e., might contain other variables. Since lower bounds must be subtypes of upper bounds, we can propagate constraints to the variables appearing in the bounds.

We illustrate our use of CHR by presenting some rules that describe the interaction of primary constraints. Our two primary constraints are

– `subTypeOf(ID,L)`: The type of identifier `ID` is a subtype of some type in `L`, where `L` is a list of polymorphic type expressions.
– `superTypeOf(ID,T)`: The type of identifier `ID` is a supertype of type `T`, a polymorphic type expression.

With polymorphic types we can restrict the domain by a type expression containing the – not yet known – type of another identifier. If the type of such an identifier becomes known, the latter is replaced by the type in the constraint. For example, consider the following two constraints:

```
subTypeOf(id(1),[float,list(id(2))])
superTypeOf(id(1),tuple([id(3),int])
```

Suppose the types of `id(2)` and `id(3)` both turn out to be `int`. Then the above two constraints are automatically replaced with constraints:

```
subTypeOf(id(1),[float,list(int)])
superTypeOf(id(1),tuple([int,int])
```

Due to the lower bound, `float` can be eliminated from the upper bound. This is performed by the following CHR rule:

```
superTypeOf(X,A) \ subTypeOf(X,B0) <=> eliminate_sub(A, B0, B) |
      subTypeOf(X, B).
```

Here, the Prolog predicate: `eliminate_sub(A,B0,B)` means that the list of upper bounds `B0` can be reduced to a proper subset `B` based on lower bound `A`.

To conclude the above example, we obtain:

```
subTypeOf(id(1),[list(int)])
superTypeOf(id(1),tuple([int,int])
```

In another example, we show how two upper bounds on the same identifier are handled. Suppose we have the following constraints:

```
subTypeOf(id(1),[float,list(int)])
subTypeOf(id(1),[tuple([int,int]),func(int,float)])
```

The upper bounds trigger the following CHR rule:

```
subTypeOf(X,T1), subTypeOf(X,T2) <=> type_intersection(T1,T2,T) |
      create_log_entry(intersection(X,T1,T2, T)),
      subTypeOf(X,T).
```

The predicate `type_intersection(T1,T2,T)` posts a constaint stating `T` is the intersection of `T1` and `T2`. We obtain a single upper bound:

```
subTypeOf(id(1),[tuple([int,int])])
```

Interaction of the Secondary Constraints Unfortunately, it is not realistic to capture all interactions of secondary constraints as that would require exponentially many rules in the number of constraints. Hence, we only handle the interaction of secondary constraints with primary constraints. This means, we do not have any CHR rules with multiple secondary constraints in their heads. Secondary constraints restrict domains by generating the proper primary and secondary constraints, when the domains of their arguments are sufficiently narrow: if certain arguments of the constraints are within a certain domain, then some other argument can be restricted further.

We obtain most of our secondary constraints from the program syntax. In general, a syntactic construct imposes restrictions on the types of its subconstructs. E.g., the type of the left side of an assignment has to be at least as "broad" as the type of the right side. Similarly, for every built-in function, there is a well-defined relation between the types of its arguments and the type of the result. These relations can be expressed with corresponding CHR constraints.

For example, we use the secondary constraint `sum/3` to capture the relation between the types of arguments and that of the result of the built-in function '+'. Let us consider the Q expression `x+y`, and let the types associated with `x`, `y` and `x+y` be `id(1)`, `id(2)`, and `id(3)`, respectively. This Q expression gives rise to the secondary constraint `sum(id(1), id(2), id(3))`. If the first argument of `sum/3` turns out to be integer, then the type of the second argument and the type of the result must be the same (according to the behaviour of the function '+' in Q). Consequently, the `sum` constraint can be removed from the constraint store, and a new constraint `eq(id(2), id(3))` is added, expressing the equivalence of two types. This is performed by the following CHR simplification rules:

```
sum(X,Y,Z) <=> known_type(X,int) | eq(Y,Z).
sum(X,Y,Z) <=> known_type(Y,int) | eq(X,Z).
```

3.3 Error Handling

During constraint reasoning, a failure of Prolog execution indicates some type conflict. In such situations, before we roll back to the last choice point, we remember the details of the error. We maintain a log[7] that contains entries on how various domains change during the reasoning and what constraints were added to the store. Furthermore, to make error handling more uniform, whenever secondary constraints are found violated, they do not lead to failure, but they set some domain empty. Hence, we only need to handle errors for primary constraints. Whenever a domain gets empty, we mark the expression associated with the domain and we look up the log to find the domain restrictions that contributed to the clash. We create and assert an error message and let Prolog fail. For example, the following message

```
Expected to be broader than (int -> numeric) and
               narrower than (int -> int)
  file:samples/s1.q  line:13  character:4
   {[x] f[x]}
   ^^^^^^^^^^
```

indicates that the underlined function definition is erroneous: the return value is numeric or broader (inferred from the type of `f`), although it is supposed to be narrower than integer (inferred from a type declaration).

3.4 Labeling

After all constraints are added to the constraint store, we use labeling to find a type assignment to each program expression (i.e., to each identifier associated with a node of the abstract syntax tree) that satisfies the constraints. This involves another traversal of the abstract syntax tree. We select the next identifier `X` to be labelled and set its domain to a singleton set, based on its current domain. We implemented this by adding a new constraint `label(X)`. This constraint triggers the narrowing of the domain of `X` through the following CHR rules:

```
label(X) <=> id_known_type(X,_) | true.
label(X), superTypeOf(X,A), subTypeOf(X,L) <=>
        label_upwards(X,A,L,Type),
        hasType(X,Type).
label(X), superTypeOf(X,A) <=>
        label_upwards(X,A,[any],Type),
        hasType(X,Type).
```

[7] Recall that we use the `create_log_entry` procedure in all CHR rules relevant to creating error messages.

```
label(X), subTypeOf(X,L) <=>
        label_downwards(X,L,Type),
        hasType(X,Type).
label(X) <=>
        label_downwards(X,[any],Type),
        hasType(X,Type).
```

First, we check if the type of `X` is already known. If so, we do nothing. Otherwise, we have four cases based on the presence or absence of a lower and upper bound:

– If we have a lower and an upper bound, we nondeterministically select a type from the domain. We start from the lower bound and successively try the broader types. This directionality is comfortable for implementation, because while a type might have many subtypes (e.g. any tuple of integers is a subtype of the type 'list of integers'), it has only few supertypes.
– If only a lower bound is present, we set the upper bound to `any` and proceed as in the previous case.
– If only an upper bound is present, we start from that type and go successively to its subtypes.
– If there is neither a lower, nor an upper bound, then we assume an implicit upper bound `any` and proceed as above.

Note that the `hasType/2` constraint, use above in the labeling code, translates to an upper and a lower bound:
`hasType(X,Y):- subTypeOf(X,[Y]), superTypeOf(X,Y).`

4 Difficulties

In this section, we discuss some difficulties that we had to overcome during the implementation of the type inference tool. These problems arose on one hand from some special features of the Q language, and on the other hand from some limitations of the CHR library used. We hope that these experiences can be useful for the CHR community.

4.1 Handling Meta-Constraints

As we described earlier, several built-in functions of Q have a special behaviour, called item-wise extension. We discuss the implementation of this feature now.

Let us consider, for example, the constraint `sum` which captures the relation between the arguments and the result of the built-in function '+'. If some of the arguments turn out to be lists, then the relation should be applied to the types of the list elements. We could capture this by adding adequate rules to the `sum` constraint. However, the rules describing the list extension behaviour would have to be repeated for each built-in function, which is counter-productive. Instead, we introduced a meta-constraint `list_extension/3`.

Consider a binary built-in function f, which extends item-wise to lists in both arguments and which imposes constraints `Cs` on its atomic arguments and result. Suppose that f has argument types identified by `X`, `Y` and a result type identified by `Z`. We cannot add the constraints of `Cs` to the constraint store until we know that the arguments are all of atomic type. Instead, we use the meta-constraint `list_extension(Dir,Args,Fun)`, where `Dir` specifies which arguments can be extended item-wise to lists, `Args` is the list of arguments on which the list of constraints[8] imposed by function `Fun`, will have to be formulated.

Hence, the constraint `list_extension(both,[X,Y,Z],+)` is added in our example. If later the input arguments are inferred to be atomic, then the meta-constraint `list_extension/3` adds the atomic constraints `Cs` and removes itself:

```
subTypeOf(X,Ux), subTypeOf(Y,Uy) \
  list_extension(both,[X,Y,Z],Fun) <=> nonlist(Ux), nonlist(Uy) |
  list_ext_constraints(Fun,[X,Y,Z],Cs), ( foreach(C,Cs) do C ).
```

Here, the complicated part is to find the arguments of the proper constraints imposed by the given built-in function. We solved this by asserting the relevant information in the `list_ext_constraints` predicate. E.g. in the case of the Q function '+' we have the following fact:

```
list_ext_constraints(+, [A,B,C], [sum(A,B,C)]).
```

If, on the other hand, some argument turns out to be a list, the meta-constraint is replaced by another one. For example, if we know that the types of `X` and `Y` are `list(A)` and `list(B)`, then the type of `Z` must be a list as well and we replace the `list_extension` constraint with the following two constraints: `list_extension(both,[A,B,C],+)` and `hasType(Z,list(C))`.

In fact, the difficulty of the implementation was caused by the following restriction of CHR: it is not possible to refer to a constraint in a rule head by supplying a variable holding its name and a list of its arguments (cf. the `call/N` built-in predicate group of Prolog).

To express item-wise extension, it would be more convenient to write rules where the name of a constraint can also be a variable. If such "meta-rules" were available the `list_extension` meta-constraint would become unnecessary.

For example, in the case of unary functions, where the corresponding constraint has two arguments (the input and the output types), item-wise extension could be implemented using the following, quite natural "meta-rule"[9]:

```
call(Cons,A,B) <=> is_list(A,X), is_list_extensible(Cons) |
      call(Cons,X,Y), hasType(B,list(Y)).
```

where `is_list_extensible(Cons)` succeeds exactly when `Cons` has the list-extension behaviour, `is_list(A,X)` means that the type of `A` is `list(X)`.

[8] Note that there are several built-in functions, whose type is described using more than one constraint.

[9] Here we assume that CHR supports meta-constraints in rule heads using the `call/N` formalism of Prolog.

4.2 Copying Constraints over Variables

Local variables are made globally unique by the parser. This means, that variables with the same name have the same value, so we can constrain their types to be the same. However each occurrence of a variable that holds a polymorphic function can have a different type assigned. Let us show an example:

```
f:{[x] x+2}                    (1)
...
f [2]                          (2)
...
f [1.1f]                       (3)
```

In the first line, `f` is defined to be a function having a single argument `x` which returns `x+2`. This means that the type of `f` is a (polymorphic) function which maps `A` to `B` (`A -> B`), where a secondary constraint `sum(A, int, B)` holds between the argument and the result. In (2) and (3) there are two different applied occurrences of function `f`, which specialise this `sum` constraint in two independent ways. In these examples `f` is applied to an integer and to a float, therefore the types of the second and third occurrence of `f` are `int -> int` and `float -> float`.

The above example shows that if the type of a variable is a (polymorphic) function then we cannot assume that the type of an applied occurrence is the same as that of the defining occurrence. To capture the relationship between these types we introduced a relationship, called "specialisation", which holds if the type of the applied occurrence can be obtained from that of the defining occurrence by first copying it and then substituting zero or more type variables in it with (possibly polymorphic) types.

A straightforward natural implementation of the "specialisation" constraint would be the following:

– at the defining occurrence of a variable: post the relevant type constraints;
– at the applied occurrence of a variable: read the type constraints posted for a variable and apply the "specialisation" relationship.

This approach requires that the CHR library provides means for accessing the constraints that involve a specified argument, a feature similar to the `frozen(X. Goals)` built-in predicate of SICStus Prolog. Unfortunately, the CHR implementations we used do not have this feature. This means that a Q variable holding a polymorphic function has to be treated specially: the constraints involving its type have to be collected and remembered, so that they can be accessed at the applied occurrences of the given Q variable.

We strongly believe that in order to support this and similar use cases, CHR libraries should provide access to the constraints that are in the store.

4.3 Handling Equivalence Classes of Variables

The constraint system yields lots of equalities. For example, two occurrences of the same (non-function-valued) variable give rise to an equality constraint.

One way to handle this is to propagate all primary constraints between equal variables, i.e. whenever $X = Y$, Y inherits all primary constraints of X and the other way round. For example, a simple implementation of propagating the upper bounds in the equality constraint (`eq/2`) would be the following:

```
eq(X,Y), subTypeOf(X, T) ==> subTypeOf(Y,T).            (1)
eq(X,Y), subTypeOf(Y, T) ==> subTypeOf(X,T).            (2)
```

Unfortunately, this solution is rather inefficient, since all reasoning is repeated for each variable made equal to some other one. Moreover, we have found cases which lead to an infinite propagation of CHR constraints. In the following paragraph we outline an example of this.

As we have seen in Section 3 two upper bounds on a variable are replaced with their intersection. Let us suppose that variable `A` has two upper bounds `list(X)` and `list(Y)`. There is an *intersection rule* which replaces these two with the upper bound `list(Z)`, where `Z` is a new variable and `Z` $\leq$ `X` and `Z` $\leq$ `Y` also have to be satisfied. Consider the following state of the constraint store:

```
eq(id(1), id(2)),
subTypeOf(id(1), [list(id(3))]),
subTypeOf(id(2), [list(id(4))]).
```

First the equality rule can fire, yielding two upper bounds on `id(1)` and `id(2)`. Now, the intersection rule can produce new upper bounds on these variables, which can be propagated to the other variable by the equality rule again.

It is easy to show that the above constraint store yields an infinite loop using these rules. Consider the following condition C: The two variables (`id(1)` and `id(2)`) have got at least two upper bounds in total, where each variable has at least one, and there exist two upper bounds, on which the intersection rule has not fired yet. It is easy to see that if C holds, then at least one rule can fire (intersection, or equality). On the other hand C is an invariant condition, as when any of these two rules fire. C remains true, if it was true before. Together with the initial state, where C also holds, this constraint store yields an infinite loop (regardless of the rule execution order).

The problem is caused by repeating the reasoning at each equal identifier. We solved this by introducing a directionality to the constraint propagation: we take a strict total order on identifiers and only propagate constraints towards the smaller identifier. The smallest in a set of equal identifiers thus represents the whole set in the sense that it accumulates all constraints.[10] Once the type of the smallest identifier becomes known, it gets propagated back to the other identifiers. Hence, instead of `eq(X,Y)` we introduced the constraint `represented_by(X,Y)`, where `Y` $\leq$ `X` holds. Furthermore, for all constraints C we have a new rule, which states that if `X` is represented by `Y` and `X` occurs in C, then it should be substituted with `Y`. As we could not formulate meta-constraints with CHR, we had to provide propagation rules for every single constraint. For example, in case of the constraint `sum` we needed the following code:

[10] This is similar to how Prolog handles the unification of two variables.

```
represented_by(A,B) \ sum(A,C,D) <=> sum(B,C,D).
represented_by(A,B) \ sum(C,A,D) <=> sum(C,B,D).
represented_by(A,B) \ sum(C,D,A) <=> sum(C,D,B).
```

This yielded lots of new rules, however, it was easy to generate them automatically, using a small Prolog program.

There are efficiency problems even with this solution. Suppose we have the following constraint: `c(...,id(2),...)` and a propagation rule R, whose head matches the above constraint (possibly involving other heads) and the body of the rule contains a new CHR constraint: `d(...,id(2),...)`. If `id(2)` later turns out to be equivalent to `id(1)`, then we substitute `id(2)` with `id(1)` in every constraint that contains `id(2)`. This yields a store with constraints:

```
c(...,id(1),...)
d(...,id(1),...)
```

The propagation rule (R) can fire now, which might infer the second constraint (`d`) again. In order to avoid further efficiency losses we added idempotency rules for every constraint, that is, we remove duplicate constraints.

However, this solution also has a negative consequence. It is possible that duplicate constraints yield redundant inferences, if these are fired before the idempontency rules. Consider the following example: let the constraint store contain constraints C_i for all $1 \leq i \leq n$, furthermore let us suppose we have propagation rules (R_j): C_j => C_{j+1} for all $1 \leq j \leq n-1$. Let us examine what happens when C_1 is a constraint inferred redundantly (twice), as described above. If the R_j rules are fired before the idempontency rules, then it is possible that we infer all C_i constraints twice, before eliminating the duplicates. This results in $2n$ inference steps instead of the optimal 1 (if the duplicate C_1 is eliminated before applying rules R_j). The problem occurs because we have no control over the firing order of CHR rules with different heads. We believe that a way to prescribe the order of such rules, e.g. using some priorities, would often help in improving the efficiency of CHR applications.

The solution of this problem of redundant inference is still an open question.

4.4 Labeling

The implementation of labeling posed several challenges. We noticed that the order in which identifiers are selected is crucial for efficiency. For example, it is important to label subexpressions first and then find the type of a complex expression. Another example is the function application, where labeling should first assign a type to the input and then the type of the output is typically automatically inferred by the constraints. Consequently, labeling involves a traversal of the abstract syntax tree, and at each node we decide the order in which expressions are labelled based on the syntactic construct involved. Often we had to rely on heuristics as it was hard to guess what order would work best in practice.

The next difficulty arises when we already know which identifier to label, and we have to choose a value. The set of all types is infinite, so we cannot try all values for a variable during labeling. hence we made some restrictions. First, we only allow a fixed increase in the term depth of types. This depth increase was experimentally set to two. E.g. if X is known to be a subtype of `list(any)`, then we replace `any` with terms of depth at most two. Hence, we will *not* replace `any` with `list(list(list(int)))`.

Second, we restrict using the `tuple` type. This is needed because tuples can have arbitrarily many arguments. If there is an identifier X and neither its lower nor its upper bound contains the `tuple` type, then we do not assign a tuple type to it. E.g. if X has the upper bound `list(int)`, then we only try `list(int)`. If, however, X also has a lower bound `tuple([int,int])`, then we try both `tuple([int,int])` and `list(int)`. We have found no Q programs where these restrictions led labeling astray: not finding an existing assignment. This is because nested types are not typical in Q and because our constraint system tends to recognise the need for a `tuple` type before labeling.

The main challenge of labeling comes from the fact that it aims to traverse a huge search space. The abstract syntax tree can have many nodes even for moderately long programs, hence we have many identifiers. Besides, Q programs are typically full of ambiguous expressions (in terms of type), so without labeling, very few types are known for sure. All this amounts to labeling being the bottleneck of type inference.

A solution to this problem would be to find a good partitioning of the program, such that not all the tree is labelled together, but in smaller portions. Consider, for example, two function definitions. The first expression contains an expression E_1 that allows many different types. Labeling assigns one possible type to E_1 and then starts labeling the second function definition. Suppose the second definition contains a type error at expression E_2 which leads labeling to failure. Hence, we backtrack to the choice point at E_1, and assign another possible type to E_1. However, this type has nothing to do with the type mismatch – since it occurs in a different function definition, – and we get failure again at E_2. This cycle is repeated until all possible types for E_1 are tried and only then do we conclude that the contains a type error. This procedure could be made more efficient by placing a cut after labeling the first function definition, thus eliminating the irrelevant choice point. Realizing that the types of expressions in one piece of code are independent from those of another can lead to much smaller fragments to be labelled, which has the potential to drastically reduce the time spent on labeling. Dependency analysis ([1]) could be used to find a code partitioning. Also, some kind of intelligent backtracking ([2]) algorithm could be used to avoid unnecessary choice points. However, adapting these techniques to the Q language requires further work.

5 Evaluation

In this section we discuss our motivation for using CHR in the implementation of the type inference tool and summarise our experiences.

5.1 Why Use CHR?

As we have seen in Subsection 2.2, our types are not necessarily disjoint (e.g. list and tuple). If the type of an expression becomes known to be a list of integers (`list(int)`), it is possible that later it is further narrowed down to a specific tuple (e.g. `tuple(int,int)`). Such a behaviour would be quite difficult to achieve in a unification based (purely Prolog) setup. This recognition led us to handle the problem of type inference as a CSP. Nevertheless, the number of possible types – even with some depth limit – is so large that it is hard to imagine an efficient implementation based on the CLP(FD) library. Furthermore, mapping the types to natural numbers (required by most CLP(FD) libraries) is also a non-trivial task. Choosing CHR for type reasoning seemed to be a good decision, as it is flexible enough to handle the above problems. These considerations motivated the use of the CHR library.

5.2 Our Experiences with Using CHR

CHR has proved to be a good choice as it is a very flexible tool for describing the behaviour of constraints. In CHR, arbitrary Prolog structures can be used as constraint arguments, therefore it was natural to handle the special domain defined by the type language.

However, we also had negative experiences with CHR. As described in Section 4, it often would be more convenient if we could write "meta-rules" in CHR. The need to access the constraint store also arose in some situations. For efficiency reasons, we believe it would often be useful to be able to influence the firing order of rules with different heads. Furthermore, the most of the debugging of our CHR programs was seriously hampered by the lack of a tracing tool.

Conclusion

This paper summarised our experiences using CHR for type analysis of programming languages. We found CHR to be a valuable tool, however, we believe there is still room for improvement: giving the programmer greater control over the constraint reasoning mechanism could further increase programmer productivity.

Acknowledgements

The work reported in the paper has been developed in the framework of the project "Talent care and cultivation in the scientific workshops of BME" project. This project is supported by the grant TÁMOP - 4.2.2.B-10/1-2010-0009

References

1. Austin, T.M., Sohi, G.S.: Dynamic dependency analysis of ordinary programs. SIGARCH Comput. Archit. News 20(2), 342–351 (Apr 1992), `http://doi.acm.org/10.1145/146628.140395`

2. Baker, A.B.: Intelligent backtracking on constraint satisfaction problems: Experimental and theoretical results (1995)
3. Csorba, J., Szeredi, P., Zombori, Z.: Static Type Checker for Q Programs (Reference Manual) (2011), http://www.cs.bme.hu/~zombori/q/qtchk_reference.pdf
4. Fruehwirth, T.: Theory and Practice of Constraint Handling Rules. In: Stuckey, P., Marriot, K. (eds.) Journal of Logic Programming. vol. 37(1–3), pp. 95–138 (October 1998)
5. Schrijvers, T., Demoen, B.: The K.U.Leuven CHR system: implementation and application. In: First Workshop on Constraint Handling Rules: Selected Contributions. pp. 1–5 (2004)
6. SICS: SICStus Prolog Manual version 4.1.3. Swedish Institute of Computer Science (September 2010), `http://www.sics.se/sicstus/docs/latest4/html/sicstus.html`
7. Wielemaker, J.: SWI-Prolog Reference Manual. University of Amsterdam (August 2011), `http://www.swi-prolog.org/download/stable/doc/SWI-Prolog-5.10.5.pdf`
8. Zombori, Z., Csorba, J., Szeredi, P.: Static Type Inference for the Q language using Constraint Logic Programming. In: 28th International Conference on Logic Programming (ICLP 2012). Leibniz International Proceedings in Informatics (LIPIcs) (2012)