Static Type Inference for the Q language using Constraint Logic Programming

ZSOLT ZOMBORI, JÁNOS CSORBA AND PÉTER SZEREDI

Department of Computer Science and Information Theory
Budapest University of Technology and Economics
Budapest, Magyar tudósok körútja 2. H-1117, Hungary
E-mail: {zombori,csorba,szeredi}@cs.bme.hu

submitted 25 March 2012; revised ? ? ?; accepted ? ? ?

Abstract

We describe an application of Prolog: a type inference tool for the Q functional language. Q is a terse vector processing language, a descendant of APL, which is getting more and more popular, especially in financial applications. Q is a dynamically typed language, much like Prolog. Extending Q with static typing improves both the readability of programs and programmer productivity, as type errors are discovered by the tool at compile time, rather than through debugging the program execution.

We map the task of type inference into a constraint satisfaction problem and use constraint logic programming, in particular the Constraint Handling Rules extension of Prolog. We determine the possible type values for each program expression and detect inconsistencies. As most built-in function names of Q are overloaded, i.e. their meaning depends on the argument types, a quite complex system of constraints had to be implemented.

KEYWORDS: logic programming, types, static type checking, CSP, CHR, Q language

Introduction

Our paper presents most recent developments of the `qtchk` type analysis tool, for the Q vector processing language. The tool has been designed in a collaborative project between Budapest University of Technology and Economics and Morgan Stanley Business and Technology Centre, Budapest. We described our first results in (Zombori et al. 2011). That version provided *type checking*: the programmer was expected to provide type annotations (in the form of appropriate Q comments) and our task was to verify the correctness of the annotations. Since then, we moved from type checking towards *type inference*: we devised an algorithm for inferring the possible types of all program expressions, without relying on user provided type information. Our preliminary results with the type inferencer were presented in (Csorba et al. 2011). Now we report on the more mature `qtchk` system that is nearly complete.

The main goal of the type inference tool is to detect type errors and provide detailed error messages explaining the reason of the inconsistency. Our tool can help detect program errors that would otherwise stay unnoticed, thanks to which

it has the potential to greatly enhance program development. Although we no longer require type annotations, we allow them as they provide documentation and improve maintenance and code reuse. We believe that the type language devised in the framework of this project is a comfortable language for documenting types.

We perform type inference using constraint logic programming: the initial task is mapped into a constraint satisfaction problem (CSP), which is solved using the Constraint Handling Rules extension of Prolog (Fruehwirth 1998), (Schrijvers and Demoen 2004).

Our paper is structured as follows. In Section 1 we give some background information that will be important for understanding the rest of the paper. Afterwards, in Section 2 we overview approaches to type inference that are related to our work. Section 3 contains our main contribution: we present static type inference as a constraint satisfaction problem and address the theoretical difficulties that this approach yields. Section 4 presents the `qtchk` program, a static type inferencer for Q that implements the algorithm outlined in Section 3. Due to lack of space, we only address type inference proper. More details about parsing Q programs and the system architecture can be found in (Zombori et al. 2011). In Section 5 we evaluate our tool. We present test results and report on our experiences with porting the program from SICStus to SWI Prolog. Finally, we end with a brief conclusion.

1 Background

In this section we present the Q programming language. Due to lack of space we do not describe the necessary background related to constraint logic programming: we expect the readers to be familiar with the constraint satisfaction problem, the Prolog language and the Constraint Handling Rules (CHR) extension of Prolog.

1.1 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. Consequently, numerous investment banks (Morgan Stanley, Goldman Sachs, Deutsche Bank, Zurich Financial Group, etc.) use this language for storing and analysing financial time series[1]. The Q language (Borror 2008) first appeared in 2003 and is now (February 2012) so popular, that it is ranked among the top 50 programming languages by the TIOBE Programming Community (TIOBE 2010).

Types Q is a strongly typed, dynamically checked language. This means that while each variable 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

[1] Kx-Systems: `http://kx.com/Customers/end-user-customers.php`

and time related types that facilitate time series calculations. Q has the following 16 atomic types: `boolean`, `byte`, `short`, `int`, `long`, `real`, `float`, `char`, `symbol`, `date`, `datetime`, `minute`, `second`, `time`, `timespan`, `timestamp`.

- **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. For example, `((‘a;‘b) ! (1;2))` is a dictionary that maps symbols `a,b` to integers `1,2`, respectively.
- **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. Input and return values of functions can also be functions: for example, a special group of functions, called *adverbs* take functions and return a modified version of the input.

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.

Type restrictions in Q The program code environment can impose various kinds of restrictions on types of expressions. In certain contexts, only one type is allowed. For example, in the do-loop `do[n;x:x*2]`, the first argument specifies how many times `x` has to be multiplied by `2` and it is required to be an integer. In other cases we expect a polymorphic type. If, for example, function `f` takes arbitrary functions for argument, then its argument has to be of type `A -> B` (a function taking an argument of type `A` and returning a value of type `B`), where `A` and `B` are arbitrary types. In the most general case, there is a restriction involving the types of several expressions. For instance, in the expression `x = y + z`, the type of `x` depends on

[2] Assignment is denoted by a colon, e.g. `x:x*2` doubles the value of the variable `x`.

those of `y` and `z`. A type analyser for Q has to use a framework that allows for formulating all type restrictions that can appear in the program.

1.2 Restriction of the Q language for type reasoning

Q is a very permissive language. In consultation with experts at Morgan Stanley we decided to impose some restrictions on the language supported by the inference tool, in order to promote good coding practice and make the type analyser more efficient.

With multiple assignment variables and dynamic typing, Q allows for setting a variable to a value of type different from that of the current value. However, this is not the usual practice and it defies the very goal of type checking. Hence we agreed that each variable should have a single type in a program, otherwise the type analyser gives an error message.

Other restrictions concern the types of the built-in functions. Most built-in functions in Q are highly overloaded, thanks to which some functions do not raise errors for certain "strange" arguments. For example, the built-in function `last` takes a list as argument and returns the last element of the list. However, this function works on atomic arguments as well: it simply returns the input argument. To increase the efficiency of the type reasoner we decided to ignore some special meanings of some built-in functions. For example, we neglected this special meaning of the `last` function. Consequently, we infer that the argument of the `last` function is a list, which is not necessarily true in general.

2 Related Work

One of the first algorithms used for type inference is the Hindley-Milner type system (Hindley 1969). It associates the program to be analysed with a set of equations which can be solved by unification. It supports parametric polymorphism, i.e., allows for using type variables. The type inferred by the algorithm for an expression is guaranteed to be the most general possible type, the *principal type*. Most type systems for statically typed functional languages can be seen as extensions of the Hindley-Milner system. Some of the best known examples are the ML family (Pottier and Remy 2005) and Haskell (Jones 1999). We also find several examples of dynamically typed languages extended with a type system allowing for type checking and type inference. These attempts aim to combine the safeness of static typing with the flexibility of dynamic typing. (Mycroft and O'Keefe 1984) describe a polymorphic type system for Prolog, which is essentially the same as that of ML. Here, the only addition to the language are type declarations, and it is guaranteed that any well-typed program will behave identically with or without type analysis.

A major limitation of the Hindley-Milner system is that it requires disjoint types. In such a system one cannot have, for example, a *numeric* and an *integer* type since they are not disjoint. Another approach to type inference with does not suffer from this limitation is based on subtyping (Cardelli and Wegner 1985). Here, the input program is mapped into type constraints of the form $U \subseteq V$ where U and V are

types, as opposed to Hidley-Milner systems where we obtain constraints of the form $U = V$. Subtyping systems can be seen as generalisations of Hindley-Milner systems. (Marlow and Wadler 1997) present a type checker for Erlang, a dynamically typed functional language, based on subtyping. Several of the shortcomings of this system were addressed in (Lindahl and Sagonas 2006). Their tool aims to automatically discover hidden type information, without requiring any alteration of the code. The inferred types enhance program maintenance and reuse by helping programmers understand code written long ago. They introduce the notion of success typing: in case of potential type errors (for example, because a variable can have two possible types during execution and one leads to abnormal behaviour), they assume that the programmer knows what he wants. They only reject programs where the type error is certain, i.e., when there is no way the program can run correctly.

The Q language is similar to Erlang in that they are both dynamically typed functional languages. The usage of the language naturally yields many constraints of the form $U \subseteq V$ for types U, V. Still, a type system based on subtyping is not sufficient. Due to built-in functions being highly overloaded (ad-hoc polymorphism), we need tools to formulate and handle versatile and complex constraints. Constraint logic programming seems ideal for this task.

(Demoen et al. 1998) report on using constraints in type checking and inference for Prolog. They transform the input logic program with type annotations into another logic program over types, whose execution performs the type checking. They give an elegant solution to the problem of handling infinite variable domains by not explicitly representing the domain on unconstrained variables. The way variable domains are represented in the Q type inference tool was motivated by their work. (Sulzmann and Stuckey 2008) describe a generic type inference system for a generalisation of the Hindley-Milner approach using constraints, and also report on an implementation using Constraint Handling Rules. The CLP($\mathcal{SET}$) (Dovier et al. 2000) framework provides constraint logic reasoning over sets. Our solution has many similarities to CLP($\mathcal{SET}$) as types can be easily seens as sets of expressions. The main difference is that we have to handle infinite sets.

3 Type Inference as a Constraint Satisfaction Problem

In this section we give an overview of our approach of transforming the problem of type reasoning into a CSP. Type reasoning starts from a 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 (atomic expressions, certain built-in functions), while other types might be provided by the user (through a type declaration, more details in Subsection 3.1). Besides, the program syntax 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 subexpression 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. However, our task is more difficult than a classical CSP, because there are infinitely many types, which cannot be represented explicitly in a list. Representing infinite domains is a challenge for performing type reasoning (details in Subsection 3.2).

3.1 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 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 can be 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 allow for defining polymorphic type expressions, such as `list(A) -> A` 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 whose elements are 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 a domain list (TE_1) and a range list(TE_2) via positional correspondence. For example, the dictionary `(‘name;‘date) ! (‘Joe; 1962)` has type `dict(tuple([symbol(name),symbol(date)]),tuple([symbol(Joe),int]))`[3].

`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.

Type Declarations Type declarations are not obligatory, however, the user can still provide a type annotation for an arbitrary expression. Such annotations appear as Q comments and hence do not interfere with the Q compiler. A type declaration

[3] To facilitate type inference for tables, we include detailed information on the domain/range of a dictionary in its type. (A record is a dictionary with the domain being a list of column names.)

gets attached to the smallest expression that it follows immediately. For example, in the code `x + y //$: int` variable `y` is declared to be an integer. There are two kinds of type declarations (declarative and imperative) with slightly different semantics – for details see (Csorba et al. 2011).

3.2 Domains

Type expressions can be arbitrarily embedded into each other (e.g. `list(int)`, `list(list(int))`, etc.). Furthermore, tuples can be of arbitrary length. Consequently, we have infinitely many types, which yields a problem that is slightly more complicated than classical CSP, since we cannot represent domains explicitly as finite lists.

Types, determined by the type language are not disjoint, for example a simple expression `1.1f` might have type `float` or `numeric` as well. It is evident that every expression which satisfies type `float` also satisfies type `numeric`, i.e., `float` is a *subtype* of `numeric`. We will use the subtype relation to represent infinite domains finitely as intervals: a domain will be represented with an upper and a lower bound.

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 expressions that satisfy T_1 also satisfy 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 expression that satisfies `tuple([int,int])` also satisfies `list(int)`, i.e., `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 we have two upper bounds L_1 and L_2 on the same variable X_i, this means the value of X_i expression has to be in $\bigcup(T_{1j} \bigcap T_{2k})$, for all $1 \leq j \leq n_1$ and $1 \leq k \leq n_2$.

A lower bound restriction for variable X_i is a single type expression T_i, meaning that T_i is a subtype of the type 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 $T_1 \cup T_2$ has to be subtype of the type of X. We do not use lists for lower bounds and hence cannot represent the intersection of lower bounds. We

chose this representation because no language construct in Q yields a conjunctive lower bound.

With the following example we demonstrate that lower and upper bounds are natural restrictions in Q: In the code `a: f[b]` function `f` is applied to `b` and the result is assigned to `a`. Suppose the type of `f` turns out to be a map from `numeric` to `tuple([int, int])`. We can infer that the type of `b` must be at most `numeric`, which can be expressed with an upper bound. The result of `f` of `b` has the type `tuple([int,int])`, which means, that the type of `a` must be at least `tuple([int,int])`, which can be expressed with a lower bound. If later the type of `a` turns out to be `list(int)` (a list of integers) and the type of `b` to be e.g. `float`, then the above expression is type correct.

3.3 Constraints

After parsing – where we build an abstract syntax tree representation of the input program – 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). Constraints that can be used for type inference can originate from the following sources in a Q program:

Type declarations If the user gives a type declaration, the expression will be treated as having the declared type.

Built-in functions For every built-in function, there is a well-defined relationship between the types of its arguments and the type of the result. These relations are expressed by adequate constraints. For each built-in function we need to implement a constraint to describe how it is supposed to narrow domains.

Atomic expressions The types of atomic expressions are revealed already by the parser, so for example, `2.2f` is immediately known to be a `float`.

Variables Local variables are made globally unique by the parser. This means, that variables with the same name are the same, hence their types are also the same. We ensure this by equating their corresponding domains.

Program syntax Most syntactic constructs impose constraints on the types of their constituent constructs. For example, the first argument of an `if-then-else` construct must be `int` or `boolean`. Another example is the assign construct. The type of the left side has to be at least as "broad" as the type of the right side. It means the type of the right side is subtype of the type of the left side.

3.4 Constraint Reasoning

In the previous subsections, we described the constraints that can be generated from the program code to restrict the types assignable to various expressions. Now we describe how the constrains are used to infer possible types.

Constraint reasoning is based on a *production system* (Newell and Simon 1972), i.e., a set of IF-THEN rules. We maintain a *constraint store* which holds the constraints to be satisfied for the program to be type correct. We start out with an initial set of constraints. A production rule fires when certain constraints appear in the store and results in adding or removing some constraints. With CHR terminology, we say that each rule has a head part that holds the constraints necessary for firing and a body containing the constraints to be added. The constraints to be removed are a subset of the head constraints. One can also provide a guard part to specify more refined firing conditions.

The semantics of the constraints is given by describing their consequences and their interactions with other constraints. At each step we systematically check for rules that can fire. The more rules we provide the more reasoning can be performed.

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 "smart" as possible. For this, we formulated rules to describe the following interactions on primary constraints[4]:

- 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., they might contain other variables. From the fact that the lower bound must be a subtype of the upper bound, we can propagate constraints to the variables appearing in the bounds.

Secondary constraints connect different variables and restrict several domains. 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 describe (fully) the interaction of secondary constraints with primary constraints, i.e., we formulate rules of the form: if certain arguments of the constraints are within a certain domain, then some other argument can be restricted. For example, if there is a summation in Q and we already know that the arguments are numeric values, then the result must be either integer or float. If the second argument later turns out to be float, then the result must be float. At this point, there is nothing more to be inferred and the constraint can be eliminated from the store.

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, we consider the program type correct.

If the upper and lower bounds on a variable determine a singleton set, 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

[4] Concrete examples of rules will be given in Section 4.

instantiation satisfies the constraint, then the latter can be removed from the store. Otherwise, the constraint fails. The failure of a constraint can be described by a CHR rule that sets the domain of some argument to the empty set.

Error Handling As we parse the input program, we generate constraints and add them to the constraint store. The production rules automatically fire whenever they can. If some domain gets restricted to the empty set, this means that the corresponding expression cannot be assigned any type, i.e., we have a type error. At this point we mark the erroneous expression, as well as the primary constraints whose interaction resulted in the empty domain. This information – along with the position of the expression – is used to generate an error message. The primary constraints are meant to justify the error. We will provide example error messages in Section 4. Once the error has been detected and noted, we roll back to the addition of the last constraint and simply proceed by skipping the constraint. This way, the type analyser can detect more than one error during a single run.

Labeling Eventually, after all constraints have been added, we obtain a constraint store such that none of the rules can fire any more. There are three possibilities:

- There were some discovered errors. Then we display the collected error messages and terminate the type inference algorithm.
- There were no type errors found and only primary constraints remain. In this case the domains described by the primary constraints all contain at least one element. We can make a consistent type assignment to each expression by picking an arbitrary element from their respective domains. Without secondary constraints, this assignment cannot go wrong. The type analyser hence indicates that it found no type errors in the program and a solution of the CSP is a possible type assignment to each expression.
- No type errors were found, however, some secondary constraints remain. In order to decide if the constraints are consistent, we do *labeling*.

Labeling is the process of systematically trying to assign values to variables from within their domains. The assignments might wake up production rules. We might obtain a failure, in which case we roll back until the last assignment and try the next value. Eventually, either we find a type assignment to all variables that satisfies all constraints or we find that there is no consistent assignment. In the first case we indicate that there is no type error. In the second case, however, we showed that the type constraints are inconsistent, so an error message to this effect is displayed. Unfortunately, due to the potentially large size of the search space traversed in labeling, it looks very difficult to provide the user with a concise description of the error.

4 Implementation – the `qtchk` program

We built a Prolog program called `qtchk` that implements the type analysis described in Section 3. It runs both in SICStus Prolog 4.1 (SICS 2010) and SWI Prolog 5.10.5.

It consists of over 8000 lines of code[5]. Constraint reasoning is performed using Constraint Handling Rules. Q has many irregularities and lots of built-in functions (over 160), due to which a complex system of constraints had to be implemented using over 60 constraints. The detailed user manual for `qtchk` can be found in (Csorba et al. 2011) 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 inferencer. The parser builds an abstract tree (AST) representation of the code, where each node represents a subexpression. Afterwards, we traverse the AST and formulate constraints on which type inference is performed. Both phases detect and store errors, which are presented to the user.

4.1 Representing variables

All subexpressions of the program are associated with CSP variables. In case 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 variables, we do not have this information at hand. Hence, instead of variables we use identifiers `ID = id(N,Type,Error)` which consist of three parts: 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,_,_))`.

4.2 Constraint Reasoning

Constraint reasoning is performed using the Constraint Handling Rules library of Prolog. CHR has proved to be a good choice as it is a very flexible tool for describing the behaviour of constraints. Any constraint involving arbitrary Prolog structures could be formulated. 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])
```

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

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) |
      create_log_entry(eliminate_sub(X,A,B0,B)), subTypeOf(X, B).
```

Here we make use of the following Prolog predicates:

- `eliminate_sub(A,B0,B)`: The list of upper bounds `B0` can be reduced to a proper subset `B` based on lower bound `A`.
- `create_log_entry(X)`: We assert a log entry used for creating error messages.

Consequently, 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)` ensures that `T` is the intersection of `T1` and `T2`. We obtain a single upper bound:

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

4.3 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 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).

4.4 Challenges

In this subsection, we mention some difficulties that we had to overcome during the implementation of `qtchk`. We do not aim for completeness nor do we have space to fully explain our solutions.

Item-wise List Extension The item-wise extension of built-in functions required further considerations. Let us see, for example the constraint `sum` which captures the relation between the arguments and the result of the function '+'. In case some of the arguments are lists, then the relation applies to the list elements. One way to capture this is to make the `sum` constraint clever enough by providing adequate rules. However, the rules describing the list extension behaviour would have to be repeated for each built-in function, which is not productive. Instead, we introduced a meta-constraint `listextension/3`. Consider a binary built-in function `f`, which extends item-wise to lists in both arguments and which imposes constraint `c` on its atomic arguments and result. Suppose that `f` has arguments identified by X, Y and result identified by Z. We cannot add `c(`X`,`Y`,`Z`)` to the constraint store until we know that the arguments are all of atomic type. Instead, we use the meta-constraint `listextension(Dir, Args, Cons)`, where `Dir` specifies which arguments can be extended item-wise to lists, `Args` is the list of arguments on which the list of constraints `Cons` will have to be formulated. Hence, the constraint `listextension(both,[`X`,`Y`,`Z`],[c])` is added in our example. If we infer that the input arguments are atomic, then we simply add the constraint `c(`X`,`Y`,`Z`)` and remove the meta-constraint. If, on the other hand some argument turns out to be a list, we replace the meta-constraint with another one. For example, if we know that the type of X and Y are `list(`A`)` and `list(`B`)`, then the type of Z must be a list as well and we replace our `listextension` constraint with the following two constraints: `listextension(both,[`A`,`B`,`C`],[c])`, `hasType(`Z`,list(`C`))`[6].

Equality of Types The constraint system yields lots of equalities (for example, two occurrences of the same variable give rise to an equality constraint). One way to handle this is to propagate all constraints between equal variables, i.e. whenever

[6] The `hasType/2` constraint translates to an upper and a lower bound:
`hasType(X,Y):- subTypeOf(X,[Y]), superTypeOf(X,Y).`

$X = Y$, Y inherits all constraints of X and vice versa. This is not efficient, though, since then all reasoning is repeated at each equal identifier. Moreover, we have found cases where we could not avoid the infinite propagation of CHR constraints. We solved this problem by introducing a directionality to the constraint propagation: we take a strict total order on identifiers and only propagate constraints towards the smaller identifier. Hence, from a set of equal identifiers, the smallest represents the set. Once its type becomes known, it gets propagated back to the represented identifiers. As we could not formulate meta-constraints with CHR, we had to provide propagation rules for every single constraint. This yielded lots of rules, however, it was easy to generate them automatically, using a small Prolog program.

Labeling : The set of all types is infinite, which we cannot all try 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 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 try the type `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 to it a tuple type. E.g. if X has the upper bound `list(int)`, then we only try `list(int)`. If, however, X is 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: this is because nested types are not typical in Q and because our constraint system tends to recognize the need for a `tuple` type before labeling.

5 Evaluation

This section consists of two parts. First we present some test results using `qtchk` to analyse large volumes of Q programs available on the web. Afterwards, we briefly report on porting the program from SICStus to SWI Prolog.

5.1 Testing

The best way to evaluate our tool would be on Q programs developed by Morgan Stanley. However, we could not obtain such programs due to the security policy of the company. Instead, we used user contributed Q examples, publicly available at the homepage of Kx-System (Borror 2008). This test set contains several (extended) examples from the Q tutorial and other more complex programs. Table 1 summarizes our findings.

We used 128 publicly available Q programs. Of this 43 were found type correct. As explained in Subsection 1.2, we made some restrictions on the Q language, following the requirements of Morgan Stanley. 43 programs were found erroneous due to not fulfilling these restrictions. Most of the error messages arise from the same variable used with different types and from some neglected special meaning of built-in functions. We often found the case that a function is called but defined in

Table 1. *Test results.*

All	*Type correct*	*Restrictions*	*Labeling timeout*	*Type error*	*Analyser error*
128	43 (33.6%)	43 (33.6%)	32 (25%)	5 (3.9%)	5 (3.9%)

another file that was not included among the examples. In such programs the lack of information often resulted in an extremely large search space to be traversed during labeling. In 32 programs labeling could not find any solution within the given time limit (1000 sec), partly for the former reason.

We were happy to find 5 genuine errors in the test set. These are in the following programs: `run.q`[7]. `mserve.q`[8]. `oop.q`[9] `quant.q`[10] and `dgauss.q`[11]. We have found 5 programs containing some language element that our tool cannot handle well. We are in the process of eliminating these problems.

5.2 Porting from SICStus to SWI

The `qtchk` tool was developed using SICStus Prolog 4.1. Afterwards, we adapted it to SWI Prolog 5.10.5. This involved changing the name of some library predicates. For example, the SWI equivalent of the SICStus `codesio` library is the `charsio` library. This involved replacing '`codes`' with '`chars`' in each predicate name (e.g. `format_to_codes` → `format_to_chars`). We had some difficulties with loops (Schimpf 2002) which are standard in SICStus 4, but not in SWI. We used the code provided by Joachim Schimpf[12] to expand do-loops and assert adequate auxiliary predicates. However, we had to modify the code, because the new predicates were sometimes not asserted in the correct module. Apart from the trouble around loops, we found the transition rather straightforward, with few difficulties.

Conclusions

We presented an algorithm and the tool `qtchk` that can be used for checking Q programs for type correctness. We described how to map this task into a constraint satisfaction problem which we solve using constraint logic programming tools. We have found that our program is a useful tool for finding type errors, as long as the programmers adhere to some coding practices, negotiated with Morgan Stanley, our project partner. However, we believe that the restrictions that we impose on the use of the Q language are reasonable enough for other programmers as well, and our tool will find users in the broader Q community.

[7] `http://code.kx.com/wsvn/code/contrib/cburke/qreference/source/run.q`
[8] `http://code.kx.com/wsvn/code/kx/kdb+/e/mserve.q`
[9] `http://code.kx.com/wsvn/code/contrib/azholos/oop.q`
[10] `http://code.kx.com/wsvn/code/contrib/gbaker/common/quant.q`
[11] `http://code.kx.com/wsvn/code/contrib/gbaker/deprecated/dgauss.q`
[12] `http://eclipseclp.org/software/loops/index.html`

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

We also acknowledge the support of Morgan Stanley Business and Technology Centre, Budapest in the development of the Q type inferencer system. We are especially grateful to András G. Békés, Balázs G. Horváth and Ferenc Bodon for their encouragement and help.

References

Borror, J. A. 2008. *Q For Mortals: A Tutorial In Q Programming.* CreateSpace, Paramount, CA.

Cardelli, L. and Wegner, P. 1985. On understanding types, data abstraction, and polymorphism. *ACM COMPUTING SURVEYS 17,* 4, 471–522.

Csorba, J., Szeredi, P., and Zombori, Z. 2011. *Static Type Checker for Q Programs (Reference Manual).* http://www.cs.bme.hu/~zombori/q/qtchk_reference.pdf.

Csorba, J., Zombori, Z., and Szeredi, P. 2011. Using constraint handling rules to provide static type analysis for the q functional language. *CoRR abs/1112.3784.*

Demoen, B., García de la Banda, M., and Stuckey, P. 1998. Type constraint solving for parametric and ad-hoc polymorphism. In *Proceedings of Australian Workshop on Constraints.* 1–12.

Dovier, A., Piazza, C., Pontelli, E., and Rossi, G. 2000. Sets and constraint logic programming. *ACM Trans. Program. Lang. Syst. 22,* 5 (Sept.), 861–931.

Fruehwirth, T. 1998. Theory and Practice of Constraint Handling Rules. In *Journal of Logic Programming*, P. Stuckey and K. Marriot, Eds. Vol. 37(1–3). 95–138.

Hindley, R. 1969. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society 146*, pp. 29–60.

Jones, M. P. 1999. Typing Haskell in Haskell. In *Haskell Workshop.*

Lindahl, T. and Sagonas, K. F. 2006. Practical type inference based on success typings. In *PPDP*, A. Bossi and M. J. Maher, Eds. ACM, 167–178.

Marlow, S. and Wadler, P. 1997. A practical subtyping system for Erlang. *SIGPLAN Not. 32*, 136–149.

Mycroft, A. and O'Keefe, R. A. 1984. A polymorphic type system for Prolog. *Artificial Intelligence 23,* 3, 295–307.

Newell, A. and Simon, H. 1972. *Human Problem Solving.* Prentice Hall, Englewood Cliffs.

Pottier, F. and Remy, D. 2005. The essence of ML type inference. *Advanced Topics in Types and Programming Languages*, 389–489.

Schimpf, J. 2002. Logical loops. In *ICLP*, P. J. Stuckey, Ed. Lecture Notes in Computer Science, vol. 2401. Springer, 224–238.

Schrijvers, T. and Demoen, B. 2004. The K.U.Leuven CHR system: implementation and application. In *First Workshop on Constraint Handling Rules: Selected Contributions.* 1–5.

SICS. 2010. *SICStus Prolog Manual version 4.1.3.* Swedish Institute of Computer Science. `http://www.sics.se/sicstus/docs/latest4/html/sicstus.html`.

Sulzmann, M. and Stuckey, P. J. 2008. HM(X) type inference is CLP(X) solving. *Journal of Functional Programming 18*, 251–283.

TIOBE. 2010. TIOBE programming-community, TIOBE index. `http://www.tiobe.com`.

Zombori, Z., Csorba, J., and Szeredi, P. 2011. Static type checking for the q functional language in prolog. In *ICLP (Technical Communications)*, J. P. Gallagher and M. Gelfond, Eds. LIPIcs, vol. 11. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 62–72.