Static Type Checking for the Q Functional Language in Prolog

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 24 January 2011; revised ? ? ?; accepted ? ? ?

Abstract

We describe an application of Prolog: a type checking 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 designed a type description syntax for Q and implemented a parser for both the Q language and its type extension. We then implemented a type checking algorithm using constraints. 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.

Prolog proved to be an ideal implementation language for the task at hand. We used Definite Clause Grammars for parsing and Constraint Handling Rules for the type checking algorithm. In the paper we describe the main problems solved and the experiences gained in the development of the type checking tool.

KEYWORDS: logic programming, types, static type checking, constraints, CHR, DCG

1 Introduction

The paper presents a type analysis tool for the Q vector processing language, which has been implemented in Prolog. The tool has been developed in a collaborative project between Budapest University of Technology and Economics and Morgan Stanley Business and Technology Centre, Budapest.

In Section 2 we give some background information on the Q language and typing. Next, in Section 3 an overview of the type analysis tool is presented. The subsequent three sections discuss the three main tasks that had to be solved in the development of the application: extending Q with a type description language (Section 4); implementing the parser (Section 5); and developing the type checker (Section 6). In Section 7 we provide an evaluation of the tool developed and give an outline of future work, while in Section 8 we provide an overview of approaches related to our work. Finally, Section 9 concludes the paper.

2 Background

In this section we first present the Q programming language. Next, we introduce some notions related to type handling that will be important for the description of the type analyser.

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. Consequently, numerous investment banks (Morgan Stanley, Goldman Sachs, Deutsche Bank, Zurich Financial Group, etc.) use this language for storing and analysing financial time series (Kx-Systems). The Q language first appeared in 2003 and is now (December 2010) 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 and stored explicitly, but inferred from 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. 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. However, if a variable is initialised to a list of atomic values of the same type, then certain operations, e.g. updating a certain element of the list, insist on keeping the list homogeneous.
- **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 called **records**, that correspond to SQL records.

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.

As an example, consider the expression

```
f: {[x] $[x>0;sqrt x;0]}
```

which defines a function of a single argument x, returning $\sqrt{x}$, if $x > 0$, and 0

otherwise. Note that the formal parameter specification `[x]` can be omitted from the above function, as Q assumes `x`, `y` and `z` to be implicit formal parameters.

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.

While being a functional language, Q also has imperative features, such as multiple assignment variables, loops, etc.

Q is often used for manipulating data stored in tables. Therefore, the language contains a sublanguage called Q-SQL, which extends the functionality of SQL, while preserving a very similar syntax.

Besides expressions to be evaluated, a Q program can contain so called *commands*. Commands control aspects of the Q environment. Among many other tasks they are responsible for changing the current context (namespace), performing various O/S level operations, loading a file, etc.

Principles of evaluation In Q, expressions are always parsed from right to left. For example, the evaluation of the expression `a:2*3+4` begins with adding `4` to `3`, then the result is multiplied by `2` and finally, the obtained value is assigned to variable `a`. In Q it is quite common to have a series of expressions $f_1\ \ f_2$, evaluated as the function f_1 applied to arguments f_2. For example we can define the function $\sqrt[4]{x}$ using the above function for $\sqrt{x}$ as follows: `g : f f`.

There is no operator precedence, one needs to use parentheses to change the built-in right-to-left evaluation order.

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*: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 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.

2.2 Typing

Static vs. Dynamic Typing The Q language is dynamically typed. This means that the type of an expression is not explicitly associated with it, but the type is stored along the value. However, since the value is only known during execution, there is no possibility of detecting a type error during compilation. In statically typed languages, each expression has a declared type associated with it, which is known before execution. Handling type information requires extra effort, but it allows for compile time detection of some errors, namely type errors.

Type Checking vs. Type Inference Type analysis comes in two different flavours. The easier task is *type checking*, when the types of expressions are provided (typically by the programmer) and the algorithm only needs to check if the type assignments are consistent. If types are not provided, the more difficult task of *type inference* is necessary, i.e., it is the type analyser that has to infer the types of complex expressions from those of atomic expressions and from the program syntax. There can be all sorts of transitions between pure type checking and pure type inference, depending on how much is provided and how much needs to be inferred.

3 The Q Type Analyser – an Overview

In this section we first introduce the type analyser from the perspective of the user and then give a bird's eye view of the architecture of the system.

Using the Type Analyser The type analyser comes as a stand-alone executable application called `qtchk`. The code to be analysed has to be extended with a type declaration for each variable and named function, in the form of an appropriate Q comment. The program can take its input either from a script file or through the standard input. The analyser returns the list of errors and potential errors (warnings)[1], along with the code fragments where they occur. Besides, the user has the option to ask for detailed error message that shows all restrictions that resulted in the clash.

We give a small example. Let `sample.q` contain the following lines:

```
f //$: list(int) -> list(int)
      : {[x] $[count x;x;0]}
```

The function `f` checks the length of its argument `x` and if it is greater than zero, then it returns `x`, otherwise it returns `0`. The programmer has provided a Q comment[2] which declares the type of `f`: it is a function that maps lists of integers to lists of integers. However, if the input is an empty list, then the output is `0`, which is not a list of integers, but an integer. The type analyser detects the mismatch and gives an error message:

```
zombori@pnp src $ qtchk samples/sample.q

Type error: Could not find type that subsumes all: [list(int),int]
  file: samples/sample.q
  line: 2
  character: 10
        {$[count x;x;0]}
         ^^^^^^^^^^^^^^
Type check complete.
```

[1] In Section 6 we explain the difference between errors and warnings.
[2] In Q, comments start with the `//` characters and are terminated by a newline.

As we can see in the example, the analyser points to the exact location of the erroneous expression along with a description of the error.

The Architecture of the Type Analyser The type analysis can be divided into three parts:

- Pass 1: lexical and syntactic analysis
  The Q program is parsed into an abstract tree structure.
- Pass 2: post processing
  Some further transformations make the abstract tree easier to work with.
- Pass 3: type checking proper
  The types of all expressions are processed, type errors are detected.

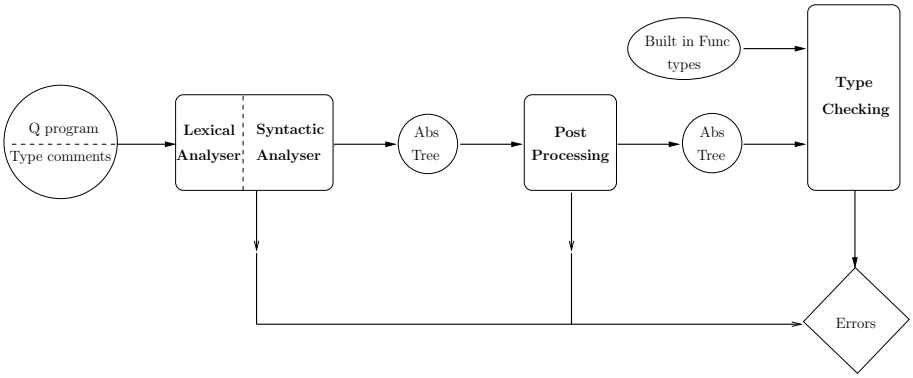

Fig. 1. Architecture of the type analyser

The algorithm is illustrated in Figure 1. The analyser receives the Q program along with the user provided type declarations. The lexical analyser breaks the text into tokens. The tokenizer recognises constants and hence their types are revealed at this early stage. Afterwards, the syntactic analyser parses the tokens into an abstract tree representation of the Q program. Parsing is followed by a post processing phase that encompasses various small transformation tasks.

In the post processing phase some context sensitive transformations are carried out, such as filling in the omitted formal parameter parts in function definitions; and finding, for each variable occurrence, the declaration the given occurrence refers to.

Finally, in pass 3, the type analysis component traverses the abstract tree and imposes constraints on the types of the subexpressions of the program. This phase builds on the user provided type declarations and the types of built-in functions. The latter are listed in a separate text file with a syntax that is an extension of the type language described in Section 4. The difference is that the current version of the type analyser does not support polymorphic types for user defined

functions, while this is unavoidable for characterising built in functions.[3] The built-in functions defined this way, it is very easy to extend the type analyser with further built-in functions, should there be new ones. The predefined constraint handling rules trigger automatic constraint reasoning, by the end of which the types of all subexpressions are inferred.

Each phase of the type analyser detects and stores errors. At the end of the analysis, the user is presented with a list of errors, indicating the location and the kind of error. In case of type errors, the analyser also gives some justification, in the form of conflicting constraints.

The type checking tool has been implemented in SICStus Prolog 4.1 (SICS 2010). The subsequent sections deal with three main parts of the development process: extending Q with a type language, implementing parsing and implementing type checking.

4 Extending Q with a Type Language

In order to allow the users to annotate their programs with type declarations, we had to devise a type language that could be comfortably integrated into a Q program. Type annotations appear as Q comments and hence do not interfere with the Q compiler. A type declaration can appear anywhere in the program and it will be 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.

Due to lack of space, we only illustrate the type language, without any pretence to completeness. The type language contains usual atomic types, such as `int, float, real, symbol` and constructors for building lists (`list(int)`), dictionaries (`dict(int,list(int))`), tables (`table(‘name: symbol; ‘age: int)`), records (`record(‘name: symbol; ‘age: int)`) and functions (`int -> symbol`). There are a couple generic types, such as `numeric` and `any`. Furthermore, we introduced some other types, such as the tuple type, which allow us to describe fixed length generic lists (`tuple(int,real,int)`). Such types require extra care, because the same expression can have different descriptions. For example `(3;2.2;4)` could have type `list(numeric)` or `tuple(int,float,int)`, and this has to be kept in mind constantly during type analysis. Type constructors can be embedded into each other, building complex types of arbitrary depth.

Type declarations can be of two kinds, having slightly different semantics: *imperative* (believe me that the type of expression E is T) or *interrogative* (I think the type of E is T, but please do check). To understand the difference, suppose the value of `x` is loaded from a file. This means that both the value and the type is determined in runtime and the type checker will treat the type of `x` as `any`. If the user gives an imperative type declaration that `x` is a list of integers, then the type analyser will believe this and treat `x` as a list of integers. If, however, the type declaration is interrogative, then the type analyser will issue a warning, because

[3] We are currently working to extend the type language to polymorphic types. See more about this in Section 7.

there is no guarantee that `x` will indeed be a list of integers (it can be anything). Interrogative declarations are used to check that a piece of code works the way the programmer intended. Imperative declarations provide extra information for the type analyser. A Q program is guaranteed to be free of type errors in case the analyser issues no errors and all the imperative declarations are indeed true.

Different comment tags have to be used for introducing the two kinds of declarations. We give an example for each:

```
f //$: int -> boolean      interrogative
g //!: int -> int          imperative
```

5 Parsing

In this section we give an outline of the data structures and algorithms used by the parser component of the type checker. The input of this phase is the original Q program extended with type declarations embedded into Q comments. Its output is the abstract syntax tree and a (possibly empty) list of lexical and syntactic errors. The parser consists of three parts: a lexical analyser, a syntax analyser and a post processor. A particular challenge of parsing Q was that there is no publicly available syntax for the language. We had to use various incomplete tutorials and experiment a lot. Not only is the language poorly documented, we found that it supports lots of exceptions and extreme cases. Some exceptions are demanded by programmers while others only lead to wrong programming habits, hence we were in constant negotiation about what the final syntax should look like.

5.1 Lexical Analysis

As the first processing step, the type checker converts its input Q program to a list of lexical elements, or tokens, for short. We defined the following kinds of lexical elements: `layout`, `string`, `identifier` (variable or built-in function), `constant` (literal value of an atom type of Q), `separator`, `graphic` (a single non-alphanumeric character that is not a separator) and `comment start`. It is the responsibility of the analyser to identify the different atomic expressions in the program, and to determine their types.

Each token contains position information, so that if a later phase discovers an error, it can point to the exact location where the error occurred. If an error occurs during the lexical analysis, a special token called *error token* is generated.

5.2 Syntactic Analysis

The syntax analyser takes a list of tokens as its input, and builds an abstract syntax tree representation of the program. The input is parsed as a sequence of Q expressions and Q commands. While most of the commands are irrelevant for the type analysis, some have to be taken into account: `\cd` changes the working directory, `\l` loads a file and `\d` controls the current context. When a load command is encountered, the referred file is parsed and included in the abstract syntax tree.

In the syntax analyser we exploit the backtracking search of the Prolog language and use the Definite Clause Grammar (DCG) (Pereira and Warren 1980) extension of Prolog to perform the parsing.

A particular challenge of building the parser was that the language syntax of Q is not publicly available. There are all sorts of online documentations that we could use, such as (Borror 2008), but the sources are all incomplete, hence we had to do lots of experiments to discover the syntax.

It is beyond the scope of this paper to present the syntax of the Q language. We only mention that the most frequent expressions are function definitions, function applications, infix function applications, list expressions, table expressions, assignments, identifiers and constant atomic values. The abstract syntax tree form of an expression consists of a node whose label identifies an expression constructor, and whose children are the abstract syntax forms of its subexpressions. For example, the abstract form of a function application is a node labelled with `app`, whose left subtree is the expression providing the function, and the right subtree is the argument (or the list of arguments). Atomic constants are represented with a node labelled `cons`,whose left subtree is the constant itself (as a Prolog string), while the right subtree identifies the type of the constant. As an example, in Figure 2 we show the abstract format of the expression `f (4+2)`.

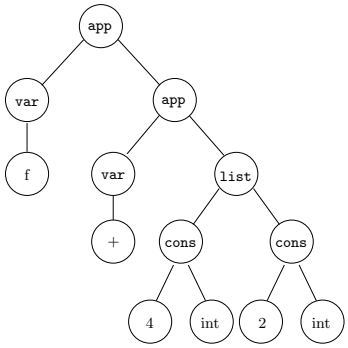

Fig. 2. The abstract tree format of the expression `f (4+2)`

In the rest of the subsection, we list some of the challenges that we had to overcome for building the syntax analyser.

Left-Recursion and Evaluation Order Our first version of the grammar was simple to understand, but it was not free from (indirect) left recursion. A formal grammar that contains left recursion cannot be parsed using DCGs. The right-to-left evaluation order of Q also caused difficulties. Both problems were solved using the well known algorithm from (Moore 2000) to convert the grammar into an equivalent right recursive grammar. The algorithm is based on dividing the problematic nonterminals to a start and a tail part. In this grammar, implementing right-to-left evaluation became simple.

Avoiding Exponential blowup Improper grammar can drive the parsing to exponential run time. Consider the following simplified grammar fragment:

```
expression :=
      ...
      |  assignment
      |  application ;
application :=
      identifier, expression ;
assignment :=
      application, :, expression ;
```

An expression starting with an application can be parsed directly as an application or as an assignment whose left-hand side starts with an application. Hence, the expression $e_1\ e_2\ \ldots e_k$ can be parsed in 2^k different ways. The problem occurs, whenever there are two disjunctive non-terminals that can be parsed to start with the same expression. We solved this problem by transforming the grammar to eliminate unnecessary choice points.

Parsing Q-SQL The query sublanguage of Q, called Q-SQL, has a syntax that differs from the rest of the code. To make matters worse, some language elements are defined both inside and outside of Q-SQL, with different meaning. For example, ',' is the join operator in Q, but it serves to separate conditional arguments in Q-SQL. The `where` function is another example. The parser had to be prepared for this double parsing, which was further complicated by the fact that one can insert a normal Q expression between parentheses into a Q-SQL expression. This requires knowledge about the current context during parsing. In our solution the DCG clauses were extended with an environment argument, which carries information about the parenthesis depth and the actual position of the parser whenever parsing inside a Q-SQL expression.

5.3 Post Processing

In the post processing phase we perform some transformations on the abstract syntax tree provided by the syntax analyser. The result is another abstract tree conforming to the same abstract syntax. This phase has two main tasks:

- Implicit formal parameters are made explicit, e.g. the function definition `f:{x+y}` is extended to `f:{[x;y] x+y}`.
- Local variables are made globally unique:
  Variable name `x` refers to different variables in different function definitions. Since we would like to associate a type with each variable, we attach a context specific part to the variable name, making all variables globally unique.

6 The Type Analysis Component

In this section we give an outline of the data structures and algorithms used by the type analysis component. The input of this phase is the abstract syntax tree, constructed by the parser. Its output is a (possibly empty) list of type errors.

6.1 Type Analysis Proper

Figure 6.1 gives a brief summary of the type analysis component. Our aim is to determine whether we can assign a type to each expression of the program in a coherent manner. Some types are known from the start, since the type analyser requires that the Q program to be checked includes type definitions for all user defined functions and variables. Furthermore, we know the types of the built-in functions. The analyser infers the types of further expressions and checks for the consistency of all types.

1. To each node of the abstract tree, we assign a type variable.
2. We traverse the tree and formulate type constraints.
3. Constraint reasoning is used to automatically
   - propagate constraints,
   - deduce unknown types
   - detect and store clashes, i.e., type errors.
4. By the end of the traversal, each node that corresponds to a type correct expression is assigned a type. The types satisfy all constraints.

Fig. 3. Overview of the type analysis component

Each expression in the concrete syntax corresponds to a subtree in the abstract syntax. Hence, we maintain a variable (in mathematical sense) for each node of the tree, that stands for the type of the subtree rooted at the node. The task of the type checker is to substitute the variables with proper types.

We use type expressions to describe the type of an expression in the Q language. The type checker uses type expressions similar to those described in Section 4, extended with type variables. In the examples below such type variables are denoted with `a`, `b`, ...

We traverse the tree and formulate context specific constraints on the type of the current node and those of its children. For instance, in the example in Figure 2, when we reach the `app` node, we know it is a function application, so the left child has to be of type `a -> b`, the right child of type `a` and the whole subtree of type `b`. In some cases the constraint determines the type of some node, but in many other it only narrows down the range of possible values. In case of clash between the restrictions, there is a type error in the program.

The type checker also detects hazardous code that contains potential type error. This is the case when the expected type of some expression is a subtype of the inferred one. An example for this is when a function is declared to expect an integer

argument and all we know about the argument is that it is numeric. We cannot determine the runtime behaviour of such a code, since the type error depends on what sort of numeric argument will be provided. Instead of an error, we give a warning in such cases that the user can decide to suppress.

Constraints are handled using the Prolog CHR (Fruehwirth 1998) library. For each constraint, the program contains a set of constraint handling rules. Once the arguments are sufficiently instantiated (what this means differs from constraint to constraint), an adequate rule wakes up. The rule might instantiate some type variable, it might invoke further constraints or else it infers a type error. In the latter case we mark the location of the error, along with the clashing constraint.

In case all variables and user defined functions are provided with a type declaration, we start the analysis with the knowledge of the types of all leaves of the abstract tree. This is because a leaf is either an atomic expression, a variable or a function. Once the leaf types are known, propagation of types from the leaves upwards is immediate, because we can infer the type of an expression from those of its subexpressions. Constraints wake up immediately when their arguments are instantiated, as a result of which the type variables of the inner nodes become instantiated. Hence, as long as sufficient type information is provided by the programmer, there is no need for labelling.

The Q program is type correct if there can be no type error during execution. In case the type checker issues neither error nor warning, the program is guaranteed to be type correct. If there are no errors but some warnings, the program is correct only if the provided type declarations are true. In this case it is up to the discretion of the programmer to decide whether to neglect the warnings.

6.2 Constraints

The constraints that can be used for type inference come from two sources. First, we know the types of atomic expressions and built-in functions. For example, `2.2` is immediately known to be a float. Similarly, we know that the function `count` is of type `any -> int`. Such knowledge allows us to set – or at least constrain – the types of certain leaves of the abstract syntax tree. The other source of constraints is the language syntax. This can be used to propagate constraints, because the language syntax imposes restrictions on the types of neighbouring nodes.

Besides these type constraints, there can be type information provided by the user at any level of the abstract tree.

Constraint Handling Rules To handle type constraints, we use constraint logic programming. More precisely we use the Prolog CHR (Constraint Handling Rules) library (Fruehwirth 1998), which provides a general framework for defining constraints and describing how they interact with each other. The advantage of CHR is that the constraint variables can take values from arbitrary Prolog structures, so we can comfortably represent all values that a type expression can have.

An Example Constraint We illustrate constraint handling with a small example. Consider the expression `x in y`, where the types of `x,y` are `X,Y`, respectively. The `in` function checks if the first argument is a member of the second. The second argument is either a list or a dictionary. The type of the whole expression is boolean and the restriction on `X,Y` is expressed using the constraint `dict_list_c(Y,_,X)`, which can be defined by the following constraint handling rules:

```
% dict_list_c(X,A,B):-
% either X is a list of type B and A is integer
% or X is a dictionary with domain type A and range type B
dict_list_c(dict(X,Y),A,B) <=> A = X, B = Y.
dict_list_c(list(X),A,B)   <=> A = int, B = X.
```

The rules remain suspended until the first argument gets instantiated to a `dict/2` or `list/1` structure. The constraint fails if the adequate types cannot be unified.

However, these rules are incomplete in two ways. First, as we have seen in Section 4, a list can also be represented as a tuple. Hence, we have to add the following rule:

```
dict_list_c(tuple(Xs),A,B)     <=>
           A = int, ( foreach(B,Xs), param(B) do true ).
```

The second problem is the lack of error handling. If the constraint fails, the whole program fails. Thus, instead of telling the user where the type error occurred, we only indicate that there is a type error, which is not useful at all. We address this by assigning an identifier to each expression. Each time an error occurs, we store the identifier and the kind of error. After all constraints exited, we retrieve the identifiers of the erroneous expressions and the relevant location in the program code. With these we can give an error message that explains the problem. The final version of the constraint handling rules for `dict_list_c/3`:

```
% dict_list_c(X,A,B):-
% either X is a list of type B and A is integer
% or X is a dictionary with domain type A and range type B
dict_list_c(X,A,B,ID) <=> nonvar(X) |
           ( X = dict(A,B)
           ; X = list(B), A = int
           ; X = tuple(Xs),  A = int,
             ( foreach(B,Xs), param(B) do true )
           ; assert(q:error(type,ID,wrong_dict_list))
           ), !.
```

The constraint wakes up as soon as the first argument is instantiated to a non-variable. Then, if it is a dictionary or a list, we can enforce the constraint by unifying some terms. If the unification succeeds, the constraint exits successfully. Otherwise, we store that an error occurred.

6.3 Issues about Type Declarations

As we have discussed in Section 3, the user is required to provide every variable and user-defined function with a type description. In this subsection we give reasons for this requirement.

As we have seen before, the immediate benefit is that the types of all leaves of the abstract tree are known at the beginning of the analysis. Without type declarations, some constraints might remain suspended and lots of types unknown. In this case we have to use some sort of labeling to assign a type to each expression.

Furthermore, if the arguments of constraints are ground, we do not have to worry about the interaction of constraints. Consider, for example the following two constraints:

```
int_or_float(X) <=> (X == int ; X == float) | true.
int_or_long(X)  <=> (X == int ; X == long)  | true.
```

If these two constraints apply to type `T`, then they will not do anything as long as `T` is a variable, even though there is only one solution, namely `T=int`. In order for the type analyser to infer this, we have to add a new rule that describes the interaction of the two constraints, such as

```
int_or_float(X), int_or_long(X) <=> X = int.
```

More complex constraints can interact in many different ways and the number of constraint handling rules necessary for capturing all interactions can be exponential in the number of constraints. Given that we work with more than 50 different constraints, it is not realistic to write up exhaustively all rules. If, on the other hand, the arguments are sufficiently instantiated that the constraints can wake up individually (not knowing about the others), then we only need to provide a couple rules for each constraint. In the above example, if `X` is instantiated, then either `X=int` and both constraints exit successfully or else at least one constraint indicates an error.

For each user defined function, we have to copy its type to each occurrence of the function. If the type is ground, copying is simple since we unify the type expressions. This, however, does not work if the type of the function contains variables that are possibly constrained. Let the type of `f` be `X -> int` where there is a constraint on `X` ensuring that it is from the set `{int, float}`. Consider the following code:

```
x:f 2
y:f 3.1
```

If we unify the type variables for each occurrence of `f` with `X -> int`, then from the first line `X` will be instantiated to `int`, which will make the type checker indicate a type error in the second line, since it will try to unify `int` with `float`. What we need is separate instances of the type of `f` with distinct variables, while holding the same constraints, which is quite complicated. Fortunately, this problem does not arise if all functions are provided with a type declaration, because the current type language does not allow non-ground type declarations. So, mandatory type declarations ensure that each function has a ground type.

7 Evaluation and Future Work

The static type checking tool has been developed in Prolog in about 6 months by the three authors of this paper. Having undergone some initial testing, it is now being evaluated on real-life Q programs at Morgan Stanley Business and Technology Centre. Even in this early stage of testing, the type checking tool pointed out several type errors in real-life Q programs.

Implementation The DCG rule formalism of Prolog was extremely useful in implementing the parser. Because no precise definition for the Q language is publicly available, the syntax accepted by the tool often changed during the development. Hence it was crucial that the parser is easy to modify. For this reason we believe that DCG was a particularly good choice.

Similarly, we were satisfied with the choice of CHR for implementing the type checking. The development of the constraint rules describing the types of the built-in functions was fairly straightforward. Even without rules describing the interaction of constraints, we experienced no performance problems in type checking (although this may change if we move on to type inference).

From Type Checking towards Type Inference In Subsection 6.3 we gave justification for requiring type information about variables and user defined functions. However, it is often rather uncomfortable for the programmer to write so many declarations, so it is worth trying to lift this restriction at least partially. First, (non-function) variables that are initialised do not require a type declaration since the analyser can infer the type from the code. For functions, in many cases it is enough to declare the types of the input parameters, since from them the type of the output can be inferred.

A related question is that of polymorphic types. The type definition language can be extended in a straightforward way to allow polymorphism. In principle, it seems to be feasible to use a variant of the Hindley-Milner algorithm (Milner 1978) for type inference involving polymorphic types. However, it is yet unclear whether this can be done with acceptable performance, given the large number of overloaded function symbols.

In the immediate future, we plan to further examine and implement all possibilities of omitting type declarations and allowing for polymorphic types.

8 Related Work

Use of Prolog for Parsing Prolog has been used for writing parsers and compilers from its very conception. (Colmerauer 1978) defines Metamorphosis Grammars as a Prolog extension to support the task of parsing, while (Pereira and Warren 1980) describe Definite Clause Grammars as a simplified form of Metamorphosis Grammars. This extension is supported by practically all Prolog implementations today.

(Cohen and Hickey 1987) give a comprehensive overview of parsing and compiling

techniques in the context of Prolog language. (Paakki 1991) reports on the Prolog implementation of a compiler for a programming language called Edison.

Types and constraints Several dynamically typed languages have been extended with a type system allowing for static type checking or type inference. (Mycroft and O'Keefe 1984) describe a polymorphic type system for Prolog. (Marlow and Wadler 1997) present a type system for Erlang, which is similar to Q in that they are both dynamically typed functional languages. Several of the shortcomings of this system were addressed in (Lindahl and Sagonas 2006). The tool presented in this work differs from ours in its motivation. It requires no alteration of the code (no type annotations) and infers function types from their usage. It does not provide type safeness, instead aims to discover provable type errors. Whenever a type error is not certain, the system will assume that it will not happen. We, on the other hand, guarantee type safety by providing warnings in cases of potential errors. Besides, type annotations are an important means to enhance program readability and although we are working to reduce the number of mandatory annotations, it is very unlikely that we would ever want to eliminate all of them. (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. We believe that their work can be useful for us as we move from type checking towards type analysis. (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.

9 Conclusions

We presented a type checking tool for the Q language as a Prolog application. We developed a type description language for the type system of Q, which helps in making Q programs easier to read and maintain. We implemented a parser, and a constraint-based type checker. Using constraints enabled us to capture the highly polymorphic nature of built-in functions due to overloading. The type checker provides type safeness: a program that is deemed type correct cannot produce type errors during execution. The tool is now being deployed in an industrial environment, with positive initial feedback.

Acknowledgements

We acknowledge the support of Morgan Stanley Business and Technology Centre, Budapest in the development of the Q type checker system. We are especially grateful to 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.

Cohen, J. and Hickey, T. J. 1987. Parsing and compiling using Prolog. *ACM Transactions on Programming Languages and Systems 9,* 2, 125–163.

Colmerauer, A. 1978. Metamorphosis grammars. In *Natural Language Communication with Computers*, L. Bolc, Ed. Lecture Notes in Computer Science, vol. 63. Springer, 133–189.

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.

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.

Kx-Systems. Representative customers . `http://kx.com/Customers/end-user-customers.php`.

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.

Milner, R. 1978. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences 17,* 3, 348–375.

Moore, R. C. 2000. Removing left recursion from context-free grammars. In *Proceedings of the first conference on North American chapter of the Association for Computational Linguistics*. 249–255.

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

Paakki, J. 1991. Prolog in practical compiler writing. *The Computer Journal 34,* 1, 64–72.

Pereira, F. C. N. and Warren, D. H. D. 1980. Definite clause grammars for language analysis – a survey of the formalism and a comparison with augmented transition networks. *Artificial Intelligence 13,* 3, 31–278.

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`.