Static Type Checking for the Q Functional Language in Prolog

Zsolt Zombori[1], János Csorba[1], and Péter Szeredi[1]

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

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.

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. We emphasize two merits of our system: 1) it enforces certain type declarations on Q programmers, making their code better documented and easier to maintain and 2) we check the type correctness of Q programs and detect type errors that can be inferred from the code before execution.

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. A more detailed version of this paper is available online as a technical communication at [16].

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 understanding 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 [4]. 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 [15].

Types Q is a strongly typed, dynamically checked language. This means that while each expression has a well-defined type, it 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. 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.
- **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 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.

Although it is 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.

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   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 few 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.

3   The Q Type Analyser – Architecture

In this section we give a bird's eye view of the architecture of the system. The type analysis can be divided into three parts:

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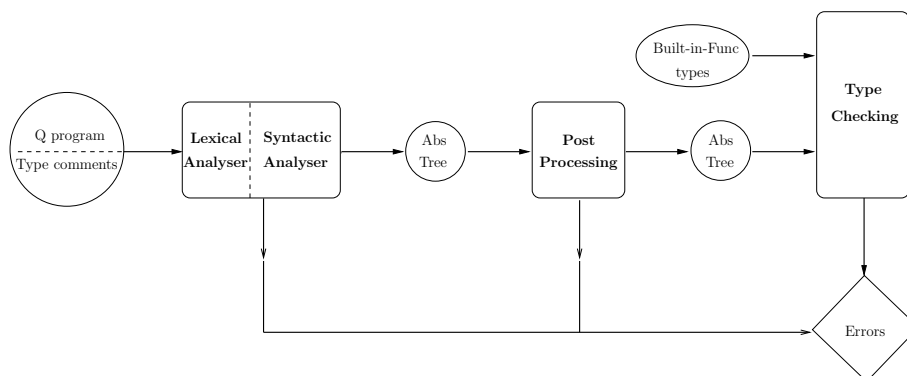
Figure 1 Architecture of the type analyser

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 syntax 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 syntax 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.[1] 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 [13]. 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 can only give a brief illustration the type language. The type language contains the atomic types of Q, 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 of 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,

---

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

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 at 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 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 with Q programmers about what the final syntax should look like.

5.1   Lexical and Syntactic Analysis

As the first processing step, the type checker converts its input Q program to a list of lexical elements, or tokens, for short. The syntax analyser takes a list of tokens as its input and builds an abstract syntax tree representation of the program. In the syntax analyser we exploit the backtracking search of the Prolog language and use the Definite Clause Grammar (DCG) [11] extension of Prolog to perform the parsing.

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)`.

In the rest of the subsection, we list some of the challenges that we had to overcome for building the syntax analyser.
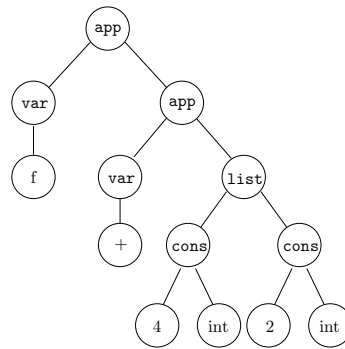
■ **Figure 2** The abstract syntax tree format of the expression `f (4+2)`

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 [8] 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.

5.2   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 syntax 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 3 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 syntax 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.

Figure 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.

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 others 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 [12] 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 syntax 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.

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 syntax tree.

Constraint Handling Rules   To handle type constraints, we use constraint logic programming. More precisely we use the Prolog CHR (Constraint Handling Rules) library [12], 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.

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 syntax 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 exhaustively write up 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 of 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.

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 [7] 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. [2] defines Metamorphosis Grammars as a Prolog extension to support the task of parsing, while [11] describe Definite Clause Grammars as a simplified form of Metamorphosis Grammars. This extension is supported by practically all Prolog implementations today.

[1] give a comprehensive overview of parsing and compiling techniques in the context of Prolog language. [10] 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. [9] describe a polymorphic type system for Prolog. [6] 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 [5]. 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. Instead of well-typing, it provides success typing: it aims to discover provable type errors. 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. [3] 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. [14] 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

---- **References** ----

1 Jacques Cohen and Timothy J. Hickey. Parsing and compiling using Prolog. *ACM Transactions on Programming Languages and Systems*, 9(2):125–163, 1987.

2 Alain Colmerauer. Metamorphosis grammars. In Leonard Bolc, editor, *Natural Language Communication with Computers*, volume 63 of *Lecture Notes in Computer Science*, pages 133–189. Springer, 1978.

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

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

5 Tobias Lindahl and Konstantinos F. Sagonas. Practical type inference based on success typings. In Annalisa Bossi and Michael J. Maher, editors, *PPDP*, pages 167–178. ACM, 2006.

6 Simon Marlow and Philip Wadler. A practical subtyping system for Erlang. *SIGPLAN Not.*, 32:136–149, August 1997.

7 Robin Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.

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

9 Alan Mycroft and Richard A. O'Keefe. A polymorphic type system for Prolog. *Artificial Intelligence*, 23(3):295–307, 1984.

10 Jukka Paakki. Prolog in practical compiler writing. *The Computer Journal*, 34(1):64–72, 1991.

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

12 Tom Schrijvers and Bart Demoen. The k.u.leuven chr system: implementation and application. In *First Workshop on Constraint Handling Rules: Selected Contributions*, pages 1–5, 2004.

13 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`.

14 Martin Sulzmann and Peter J. Stuckey. HM(X) type inference is CLP(X) solving. *Journal of Functional Programming*, 18:251–283, March 2008.

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

16 Zsolt Zombori, János Csorba, and Péter Szeredi. Static type checking for the q functional language in prolog. Technical report, Budapest, University of Technology and Economics, 2011. http://www.cs.bme.hu/ zombori/publications/2011/iclp2011/iclp2011_full.pdf.