Static Type Inference as a Constraint Satisfaction Problem

Zsolt Zombori, János Csorba, Péter Szeredi

Abstract:

We describe an application of Prolog: a type analyzer tool for the Q functional language. 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.

Introduction:

Our paper presents most recent developments of the *qtchk* type analysis tool, for the Q vector processing language. The tool has been developed in a collaborative project between Budapest University of Technology and Economics and Morgan Stanley Business and Technology Centre, Budapest. The main goal of the type analyzer 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.

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 the Prolog programming language.

In the following, we present the constraint satisfaction problem, we show how type inference can be mapped into a CSP and how the problem can be solved. Finally, we briefly report on our implementation.

Constraint Satisfaction Problem:

A constraint satisfaction problem (CSP) can be described with a triple X, D, C where

- $X = \{x_1 ... x_n\}$ is a series of variables,
- $D = \{D_1 ... D_n\}$ is a series of finite sets called domains,
- variable x_i can only take values from domain D_i,
- $C = \{c_1 ... c_k\}$ is a series of constraints, i.e., atomic relations whose arguments are variables from X.

A solution to a CSP is an assignment to each $x_i \in X$ a domain element $v_i \in D_i$ such that all constraints $c \in C$ are satisfied.

A value d_i of a variable x_i that appears in a constraint c is superfluous in case there is no assignment to the rest of the variables of c along with $x_i = d_i$ that satisfies the constraint. Removing superfluous values from the corresponding domains yields an equivalent CSP.

There are two mechanisms that lead to a solution of a CSP. First, constraints constantly monitor the domains of their variables and remove superfluous values. Second, in case constraints fail to reduce some domain to a single value, we apply labeling: we choose a variable x_i and split its domain into two parts, creating a choice point where each branch corresponds to a reduced domain. Through a backtracking search we explore the branches. Constraints can wake up as the domains of their variables change and can further eliminate superfluous values. In case a domain becomes empty, we roll back to the last choice point. By the end of labeling, either we find a single value for each variable such that all constraints are satisfied, or else we conclude that the CSP is unsatisfiable.

Mapping Type Inference into a Constraint Satisfaction Problem:

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, built-in function symbols), while other types might be provided by the user through a type declaration. Besides, the program syntax imposes restrictions that can be interpreted as constraints between the types of certain expressions. A coherent type assignment respects all user declarations and all constraints.

We associate a CSP variable with each expression of the program. Each variable has a domain, which initially is the set of all possible types. Different type restrictions can be interpreted as constraints that restrict the domains of some variables. In this terminology, the task of the analyzer 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 type expressions can be arbitrarily embedded into each other (e.g. *list(int)*, *list(list(int))*), hence there are infinitely many types, which cannot be represented explicitly in a list. Another difficulty is that the types are not necessarily disjoint. For example, the 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*. The subtype relation is a partial ordering over type expressions. We use this relation to represent infinite domains finitely as intervals: a domain is represented with an upper and a lower bound. This is possible because the type restrictions in a Q program naturally translate into upper and lower bound constraints. We demonstrate this with a simple example:

```
//$ f: numeric -> tuple([int,int])
```

a: f[b]

According to the declaration, variable **f** is a function that maps numeric values to tuples of two integers. The code that follows is a simple function application and an assignment: **f** is applied to **b** and the result is assigned to variable **a**. 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[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.

We build an abstract syntax tree representation of the input program. Afterwards, we traverse the tree and impose constraints on the types of the program expressions. The constraints describing the domains of variables 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 constrain several types and eventually restrict domains by generating primary constraints, when their arguments are sufficiently instantiated, i.e., when their domains are sufficiently narrow. The order in which constraints are added is irrelevant. Constraints that can be used for type inference can originate from the following sources in a Q program:

1. **Type declarations**: If the user gives a type declaration, the expression will be treated having the declared type.
2. **Built-in functions**: For every built-in function, there is a well-defined relation between the types of its arguments and the type of the result. These relations are expressed by adequate constraints. For each built-in function we need to implement a constraint to describe how the constraint is supposed to narrow domains.
3. **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*.
4. **Variables**: Local variables are made globally unique by the parser. This means, that variables with the same name are equal, hence their types are also equal.
5. **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 a boolean value.

Constraint Reasoning:

Constraint reasoning is based on a production system, i.e., a set of IF-THEN rules. We maintain a constraint store, i.e., the set of constraints to be satisfied for the program to be type correct. We start with the constraints generated from the abstract syntax tree. A production rule fires when certain constraints appear in the store and results in adding or removing some constraints. For example, two upper bounds on the same variable x are merged using the following rule: IF $x \leq \alpha$ and $x \leq \beta$ THEN add $x \leq \min(\alpha, \beta)$, remove $x \leq \alpha$, remove $x \leq \beta$.

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 analyzer to detect this. Hence, it is very important for the primary constraints to be as "smart" as possible. For this, we formulated rules to describe the following interactions of primary rules:

- If a variable has two upper bounds, then they should be replaced with their intersection.
- If a variable has two lower bounds, then they 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, this should be detected and the clash should be made explicit by setting the upper bounds to the empty set.
- If a variable has an upper and a lower bound that contains other variables, then adequate constraints should be added to ensure that the domain cannot reduce to the empty set.

Secondary constraints connect different variables and restrict several domains. The way they influence one variable might depend heavily on the value of some other variable(s). Hence, often secondary constraints cannot partake in the reasoning until more is known about the possible values of their arguments. Unfortunately, it is not realistic to capture all interactions of secondary constraints in our production system. For this we would need a production rule for any set of constraints such that each member has the potential to restrict the domain of the same variable. The number of rules would be exponential in the number of constraints, which is too much for any reasonably complex target language. For the Q language, we use over 60 different secondary constraints. The rules cannot be automatically generated: they are needed to capture the highly irregular nature of the language and we could not find any general pattern to characterize their interactions.

In our solution, we fully describe 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, in Q if there is a summation 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 as well. Afterwards, 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. 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.

Implementation:

We built a Prolog program called *qtchk* that implements the type analysis described in this paper. The program runs equally in SICStus Prolog 4.1 and SWI Prolog 5.10.5. It consists of over 8000 lines of code. Constraint reasoning is performed using the Constraint Handling Rules extension of Prolog. Q has many irregularities and lots of built-in functions (over 160), due to which a rather complex system of constraints had to be implemented using over 60 constraints.

Evaluation:

We have started testing on our tool. We used Q programs written by ourselves, as well as programs that can be found on the web. Here we summarize our findings.

1. Analyzing a typical Q program (100 – 200 lines of code) can take 3-5 minutes. This is slow for an interactive system, but in our case it is acceptable, since programs can be checked offline.
2. We found syntactically correct Q programs where our tool indicated a syntax error. It turned out that it was because the programs used language elements that are not needed by our partners at Morgan Stanley. We do not support the whole Q syntax, only the part that is used by Morgan Stanley.
3. We found type correct Q programs where our tool indicated a type error. This is because we make some restrictions in our type system that Q does not. These restrictions are meant to discourage dangerous coding practices and to enable more to be inferred by the tool. These restrictions are the result of negotiations with Q programmers at Morgan Stanley. The restrictions typically involve the types of built-in functions. For example, the function **raze** flattens lists of lists into a list i.e., **raze (1 2; 3 4)** results in the list **(1 2 3 4)**. When the argument of **raze** is not a list of lists, then it returns the argument unmodified. This, however, is not the intended meaning of the function and we disallow this use by declaring its type $list(list(X)) \rightarrow list(X)$.

Conclusions:

We presented the theoretical background of a tool that can be used for checking Q programs for type correctness. We proceed by mapping the initial 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. The coding practices are the ones negotiated with Morgan Stanley.

Acknowledgement (bold 14 pt): 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***