# Type Processing by Constraint Reasoning

Peter J. Stuckey[1,2], Martin Sulzmann[3], and Jeremy Wazny[2]

[1] NICTA Victoria Laboratory
[2] Department of Computer Science and Software Engineering
University of Melbourne, 3010 Australia
{pjs, jeremyrw}@cs.mu.oz.au
[3] School of Computing, National University of Singapore
S16 Level 5, 3 Science Drive 2, Singapore 117543
sulzmann@comp.nus.edu.sg

**Abstract.** Herbrand constraint solving or unification has long been understood as an efficient mechanism for type checking and inference for programs using Hindley/Milner types. If we step back from the particular solving mechanisms used for Hindley/Milner types, and understand type operations in terms of constraints we not only give a basis for handling Hindley/Milner extensions, but also gain insight into type reasoning even on pure Hindley/Milner types, particularly for type errors. In this paper we consider typing problems as constraint problems and show which constraint algorithms are required to support various typing questions. We use a light weight constraint reasoning formalism, Constraint Handling Rules, to generate suitable algorithms for many popular extensions to Hindley/Milner types. The algorithms we discuss are all implemented as part of the freely available Chameleon system.

## 1 Introduction

Hindley/Milner type checking and inference has long been understood as a process of solving Herbrand constraints, but typically the typing problem is not first mapped to a constraint problem and solved, instead a fixed algorithm, such as algorithm $\mathcal{W}$ using unification, is used to infer and check types. We argue that understanding a typing problem by first mapping it to a constraint problem gives us greater insight into the typing in the first place, in particular:

- Type inference corresponds to collecting the type constraints arising from an expression. An expression has no type if the resulting constraints are *unsatisfiable*.
- Type checking corresponds to checking that the declared type, considered as constraints, *implies* (that is has more information than) the inferred type (constraints collected from the definition).
- Type errors of various classes: ambiguity, subsumption errors; can all be explained better by reasoning on the type constraints.

Strongly typed languages provide the user with the convenience to significantly reduce the number of errors in a program. Well-typed programs can be guaranteed not to "go wrong" [22], with respect to a large number of potential problems.

N. Kobayashi (Ed.): APLAS 2006, LNCS 4279, pp. 1–25, 2006.

Typically type processing of a program either checks that types declared for each program construct are correct, or, better, infers the types for each program construct and checks that these inferred types are compatible with any declared types. If the checks succeed, the program is type correct and cannot "go wrong".

However, programs are often not well-typed, and therefore must be modified before they can be accepted. Another important role of the type processor is to help the author determine why a program has been rejected, what changes need to be made to the program for it to be type correct.

Traditional type inference algorithms depend on a particular traversal of the syntax tree. Therefore, inference frequently reports errors at locations which are far away from the actual source of the problem. The programmer is forced to tackle the problem of correcting his program unaided. This can be a daunting task for even experienced programmers; beginners are often left bewildered.

Our thesis is that by mapping the entire typing problem to a set of constraints, we can use constraint reasoning to (a) concisely and efficiently implement the type processor and (b) accurately determine where errors may occur, and aid the programmer in correcting them. The Chameleon [32] system implements this for rich Hindley/Milner based type languages.

We demonstrate our approach via three examples. Note that throughout the paper we will adopt Haskell [11] style syntax in examples.

*Example 1.* Consider the following ill-typed program:

```
f 'a' b    True = error "'a'"
f c    True z    = error "'b'"
f x    y    z    = if z then x else y
f x    y    z    = error "last"
```

Here `error` is the standard Haskell function with type $\forall a.[Char] \rightarrow a$. GHC reports:

```
mdef.hs:4:
    Couldn't match 'Char' against 'Bool'
        Expected type: Char
        Inferred type: Bool
    In the definition of 'f': f x y z = if z then x else y
```

What's confusing here is that GHC combines type information from a number of clauses in a non-obvious way. In particular, in a more complex program, it may not be clear at all where the *Char* and *Bool* types it complains about come from. Indeed, it isn't even obvious where the conflict in the above program is. Is it complaining about the two branches of the if-then-else (if so, which is *Char* and which *Bool*?), or about `z` which might be a *Char*, but as the conditional must be a *Bool*?

The Chameleon system reports:[1]

[1] The currently available Chameleon system (July 2005) no longer supports these more detailed error messages, after extensions to other parts of the system. The feature will be re-enabled in the future. The results are given from an earlier version.