Types in specification languages

Donald Sannella, Radu Grosu
dts@dcs.ed.ac.uk, grosu@informatik.tu-muenchen.de

March 14, 1996

Defining types, which imposes conceptual structure on the problem domain, seems to be a natural part of writing specifications. A rich and flexible type system obviously makes this process easier.

Although untyped languages such as the untyped λ-calculus have enormous expressive power, they are difficult to use effectively because the power is in a raw form which makes errors easy to commit and difficult to detect. Type systems are used to tame this raw power by distinguishing well-formed expressions from ill-formed ones; the latter are regarded as meaningless and are therefore rejected. The expressive power of the type system determines how many useful expressions are regarded as well-formed. Although again a rich type system that admits as many expressions as possible might seem best, there is a trade-off between tractability of typing and expressiveness of types. Another disadvantage of an overly permissive type system, for example one that "cleverly" inserts coercions automatically where required, is that expressions containing genuine errors are more likely to be accepted.

When considering powerful typing systems, it is important to distinguish between type *checking* and type *inference* (cf. type *reconstruction*). Type checking is the problem of deciding whether or not an expression is well-typed when complete typing information about all of its constituents is supplied. Type inference is the problem of finding assignments of types to constituents of an expression that makes the expression well-typed. Where more than one such assignment exists it is often important to find the "best" or "most general" type (called the *principal* type in the case of polymorphic typing systems). Type checking is in general easier than type inference. There are at least five possibilities:

1. Principal types are automatically inferrable (e.g. ML minus overloaded arithmetic operators).

2. Automatic type inference is possible but principal types do not exist in general (e.g. ML with overloaded arithmetic operators).

3. Automatic type inference is impossible in general but is possible in certain well-defined situations (e.g. inferring universe levels in ECC).

4. Automatic type inference is impossible but type checking is decidable (e.g. system F and the calculus of constructions).

5. Type checking is not decidable (e.g. most systems with subtyping, including $F_{\leq}$).

The borderline between these is rather delicate; for example, adding polymorphic recursion [Hen93] to ML takes it from class 1 to class 4. Designing a rich type system without carefully considering whether type checking will be possible is very likely to yield a system that is in class 5. (Note that type checking is always semi-decidable when the typing relation is defined by means of a system of finitary rules.) In practice, the difference between these classes is less clear-cut than it might appear because worst-case analysis does not say much about the cases that actually arise. For instance, ML type inference is DEXPTIME-complete in the worst case, but the worst case never arises in real programs and in practice ML type inference is very efficient. Similarly, there is a complete and efficient semi-decision procedure for type checking in ML with polymorphic recursion, where the non-terminating cases never show up in practice.

In a specification language it seems reasonable to allow a subtype t' to be defined by restricting a previously-defined type t to those values that satisfy a certain predicate P. (These are sometimes known as *subset types* to distinguish them from "non-logical" subtypes.) This will always make type checking undecidable unless the form of the predicate is too restricted to be useful. In such a type system, an expression e can fail to have type t' for two reasons: it might have type t but yields a value that fails to satisfy P; or it might not even have type t. In such a system we can call an expression *roughly well-typed* if it is well-typed when all the predicates are ignored. Rough type checking might be decidable when full type checking is undecidable (e.g. ASL+, see Section 7 of [SST92]). (Rough type checking is undecidable in $F_{\leq}$: there are no predicates to ignore there. Although type checking is decidable in order sorted algebra if signatures are regular [SNGM89], and it is also decidable in order sorted algebra plus retracts provided signatures are coregular [GJM85], it is not decidable whether a term in normal form would contain retracts or not.) This appears to be the best that we can hope for in such type systems, and rough type checking seems useful enough that a type system in which it is decidable should be preferred over one in which it is undecidable.

Acknowledgements: Thanks to Stefan Kahrs and David Aspinall for comments on a draft of this note.

References

[GJM85] J.A. Goguen, J.-P. Jouannaud, and J. Meseguer. Operational Semantics for Order-Sorted Algebra. In *International Conference on Automata Languages and Programming*, volume 194 of *LNCS*, 1985.

[Hen93] F. Henglein. Type inference with polymorphic recursion. *ACM Trans. on Programming Languages and Systems* 15:253–289 (1993).

[SNGM89] G. Smolka, W. Nutt, J. Goguen, and J. Meseguer. Order-Sorted Equational Computation. In *Resolution of Equations in Algebraic Structures*. Academic Press, 1989.

[SST92] D. Sannella, S. Sokołowski and A. Tarlecki. Toward formal development of programs from algebraic specifications: parameterisation revisited. *Acta Informatica* 29:689–736 (1992).