Computing Surveys 28(2):358-359, June 1996.

Type Inference for Objects

Jens Palsberg*

MIT

Type systems for object-oriented languages have been studied for more than a decade.

> **Fundamental question.** Can we design an object-oriented language and a type system such that we get all three of (1) type inference, (2) subtyping, and (3) principal types?

So far there is no satisfactory answer to this question. We can, however, obtain any two of the three items. The following is a brief examination of the concepts and of state-of-the-art.

Type inference. Types are useful because they can help a compiler generate good code and because "well-typed programs cannot go wrong." In languages such as Pascal and Ada, variables must be declared with a type, and the compiler can then verify whether a given program is indeed well-typed. This verification is called type checking. Type *inference* is the computation of missing type annotations. It allows us to omit some or all of the type annotations in our programs, and it is successfully used in functional languages such as ML and Haskell.

Subtyping. Most successful type systems for object-oriented languages have a notion of *subtyping*, i.e., an ordering of the types. This ordering is often used to allow the compiler to deduce that if an expression e has type s, and s is a subtype of t, then e also has type t. For example, if types are of the form $[l_1 : t_1, \ldots, l_n : t_n]$ where each l_i is a field name and each t_i is a type, then we may define "s is a subtype of t" to mean "s has at least the fields of t, and common fields in s and t have the same type." This definition allows an object with many fields to be used in a place where objects with fewer fields are expected.

*Lab. for Computer Science, Massachusetts Institute of Technology, NE43–340, 545 Technology Square, Cambridge, MA 02139, USA; email: palsberg@theory.lcs.mit.edu.

Principal types. In ML and Haskell, type inference produces *principal types*. A principal type summarizes all possible types of a given program fragment. This enables a modular style of type inference where we can forget the text of a program fragment once its principal type has been inferred. Principal types would be particularly useful in connection with inheritance, where the type of a derived class should only rely on the type, not the text, of its base class.

Type inference + subtyping. Palsberg [8] gave an efficient type inference algorithm that handles the form of subtyping from the example above. His algorithm is defined for an object-oriented language where each typable program fragment does not necessarily have a principal type. Mitchell and others have studied notions of constrained type, that is, a standard type together with set of subtype constraints. Mitchell [7] showed how to infer principal constrained types for λ-terms. In some cases, the principal constrained type involves a constraint set which is as big as the λ-term itself [5]. Eifrig, Smith, and Trifonov [3] showed how to infer polymorphic constrained types for classes and objects. A typable program fragment does not necessarily have a principal type in their system. For a λ-calculus, Jim [6] defined a type system with polymorphic types, intersection types, and subtyping, where each λ-term has an inferable principal typing. It is open if his approach can be applied to a language with objects.

Type inference + principal types. Rémy, Wand, Ohori and Buneman, and others have demonstrated that principal types can be inferred for a wide range of object-oriented constructs when subtyping is not considered [4, Ch.3–5].

Subtyping + principal types. When type annotations are given, type inference is the same as type checking, and "principal" is the same as "minimal in the type ordering." Cardelli and Wegner [2] introduced a typed calculus with subtyping and bounded polymorphism, and Curien and Ghelli [4, Ch.8] showed that each program fragment has a minimal type. Type checking is decidable, although not in a certain extension of the calculus, as shown by Pierce [4, Ch.12]. Cardelli [4, Ch.11] showed that many object-oriented concepts can be encoded in the calculus. Bruce's language TOOPLE [1] is object-oriented, it has a notion of subtyping, program fragments have minimal types, and type-checking is decidable.

Status. The fundamental question is still open. A satisfactory answer will improve our understanding of objects, provide better tools for existing languages, and contribute to the design of new languages.

References

[1] Kim B. Bruce, Jon Crabtree, Thomas P. Murtagh, Robert van Gent, Allyn Dimock, and Robert Muller. Safe and decidable type checking in an object-oriented language. In *Proc. OOPSLA'93, ACM SIGPLAN Eighth*

Annual Conference on Object-Oriented Programming Systems, Languages and Applications, pages 29–46, 1993.

[2] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, December 1985.

[3] Jonathan Eifrig, Scott Smith, and Valery Trifonov. Sound polymorphic type inference for objects. In *Proc. OOPSLA'95, ACM SIGPLAN Tenth Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 169–184, 1995.

[4] Carl Gunter and John Mitchell. *Theoretical Aspects of Object-Oriented Programming*. MIT Press, 1994.

[5] My Hoang and John C. Mitchell. Lower bounds on type inference with subtypes. In *Proc. POPL'95, 22nd Annual SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, pages 176–185, 1995.

[6] Trevor Jim. What are principal typings are what are they good for? In *Proc. POPL'96, 23nd Annual SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, pages 42–53, 1996.

[7] John C. Mitchell. Type inference with simple subtypes. *Journal of Functional Programming*, 1:245–285, 1991.

[8] Jens Palsberg. Efficient inference of object types. *Information and Computation*, 123(2):198–209, 1995. Preliminary version in Proc. LICS'94, Ninth Annual IEEE Symposium on Logic in Computer Science, pages 186–195, Paris, France, July 1994.