An overview of Types in Compilation*

Xavier Leroy

INRIA Rocquencourt
Domaine de Voluceau, 78153 Le Chesnay, France

1 Types in programming languages

Most programming languages are equipped with a type system that detects type errors in the program, such as using a variable or result of a given type in a context that expects data of a different, incompatible type. Such type checking can take place either statically (at compile-time) or dynamically (at run-time). Type checking has proved to be very effective in catching a wide class of programming errors, from the trivial (misspelled identifiers) to the fairly deep (violations of data structure invariants). It makes program considerably safer, ensuring integrity of data structures and type-correct interconnection of program components.

Safety is not the only motivation for equipping programming languages with type systems, however. Another motivation, which came first historically, is to facilitate the efficient compilation of programs. Static typing restricts the set of programs to be compiled, possibly eliminating programs containing constructs that are difficult to compile efficiently or even to compile correctly at all. Also, static typing guarantees certain properties and invariants on the data manipulated by the program; the compiler can take advantage of these semantic guarantees to generate better code. The "Types in Compilation" workshops are dedicated to the study of these interactions between type systems and the compilation process.

2 Exploiting type information for code generation and optimization

An early example of a type system directed towards efficient compilation is that of Fortran. The Fortran type system introduces a strict separation between integers numbers and floating-point numbers at compile-time. The main motivation for this separation, according to Fortran's designers, was to avoid the difficulties of handling mixed arithmetic at run-time [2, chapter 6]. Thanks to the type system, the compiler "knows" when to generate integer arithmetic operations, floating-point arithmetic operations, and conversions between integers and floats.

* This text is published as the introduction to the proceedings of the 1998 Types in Compilation workshop, pages 1–8, Lecture Notes in Computer Science 1473, Springer-Verlag, march 1998. http://www.springer.de/comp/lncs/index.html.

Since then, this separation has permeated hardware design: most processor architectures provide separate register sets and arithmetic units for integers and for floats. In turn, this architectural bias makes it nearly impossible to generate efficient numerical code for a language whose type system does not statically distinguish floating-point numbers from integers.

Another area where compilers rely heavily on static typing is the handling of variable-sized data. Different data types have different natural memory sizes: for instance, double-precision floats usually occupy more space than integers; the size and memory layout of aggregate data structures such as records and arrays vary with the sizes and number of their elements. Precise knowledge of size information is required to generate correct code that allocates and operates over data structures. This knowledge is usually derived from the static typing information: the type of a data determines its memory size and layout. Languages without static typing cannot be compiled as efficiently: all data representations must fit a default size, if necessary by boxing (heap-allocating and handling through a pointer) data larger than the default size — an expensive operation. Statically-typed languages whose type system is too flexible to allow this determination of size information in all cases (e.g. because of polymorphism, type abstraction, or subtyping) make it more difficult, but not impossible, to exploit unboxed data representations: see [31, 21, 34, 16, 39, 22, 33, 28] for various approaches.

Guarantees provided by the type system can also enable powerful program optimizations. For instance, in a strongly-typed language (whose type system does not allow "casts" between incompatible types), two pointers that have incompatible types cannot alias, i.e. cannot point to the same memory block. This guarantees that load and store operations through those two pointers cannot interfere, thus allowing more aggressive code motion and instruction scheduling [13]. One can also envision different heap allocation strategies for objects of different types, as exemplified by the paper by Genius *et al.* in this proceeding.

Another area where type information is useful is the optimization of method dispatch in object-oriented languages. General method dispatch is an expensive operation, involving a run-time lookup of the code associated to the method in the object's method suite, followed by a costly indirect jump to that code. In a class-based language, if the actual class to which the object belongs is known at compile-time, a more efficient direct invocation of the method code can be generated instead. If the code of the method is small enough, it can even be expanded in-line at the point of call. Simple examination of the static type of the object and of the class hierarchy of the program uncovers many opportunities for this optimization. For instance, if the static type of the object is a class C that has no subclasses, the compiler knows that the actual class of the object is C and can generate direct invocations for all methods of the object [10, 15, 5].

3 Program analyses and optimizations based on non-standard type systems

There are many points of convergence between, on the one hand, algorithms for type checking and type inference, and on the other hand, static analyses of programs intended to support code optimization. This should not come as a surprise: both static analyses and type inference algorithms attempt to reconstruct semantic information that is implicit in the program source, and propagate that information through the program, recording it at each program point. A more formal evidence is that both static analyses and type inference problems can be recast in the common framework of abstract interpretation [9]. What is more remarkable is that essentially identical algorithms are used for type inference and for certain program analyses.

For instance, unification between first-order terms, as used for type inference in the Hindley-Milner type system of ML and Haskell, is also at the basis of several fast program analyses such as Steensgaard's aliasing analysis [37], and Henglein's tagging analysis [18]. Baker [6] reflects informally on this connection between Hindley-Milner type inference and several program analyses.

Another technique that has attracted considerable interest recently both from a type inference standpoint and a program analysis standpoint consists in setting up systems of set inclusion constraints (set inequations) and solving them iteratively. This technique has been used to perform type inference for type systems with subtyping [25, 3, 14]. The same technique is also at the basis of several flow analyses for functional and object-oriented languages [35, 36, 17, 1, 32, 19, 11]. These analyses approximate the flow of control and data in the presence of first-class functions and objects, and are very effective to optimize function applications and method invocations, and also to eliminate dynamic type tests in dynamically-typed languages. Palsberg and O'Keefe [29] draw a formal connection between those two areas, by proving the equivalence between a flow analysis (0-CFA) and a type inference algorithm (for the Amadio-Cardelli type system with subtyping and recursive types). The paper by Aiken *et al.* in these proceedings surveys the use of set inclusion constraints and equality (unification) constraints for program analyses.

Several non-standard type systems have been developed to capture more precisely the behavior of programs and support program transformations. The effect systems introduced by Lucassen and Gifford [23, 20] enrich function types with effects approximating the dynamic behavior of the functions, such as input-output or operations on the store. This information is useful for code motion and automatic parallelization. Jouvelot, Talpin and Tofte [38, 40] use region annotations on the types of data structures and functions to determine aliasing and lifetime information on data structures. The ML compiler developed by Tofte *et al.* [8] relies on these lifetime information for managing memory as a stack of regions with compiler-controlled explicit deallocation of regions instead of a conventional garbage collector. Tolmach's paper in these proceedings presents a reformulation of simple effect systems as monadic type systems. Shao and Trifonov's paper develop a type system to keep track of the use of first-class continuations in a program, thus allowing interoperability between languages that support `callcc` and languages that do not.

Finally, non-standard type systems can also be used to record and exploit the results of earlier program analyses. For instance, Dimock *et al.* [12] and Banerjee [7] develop rich type systems that capture and exploit the flow information produced by flow analyses. Another example is Thiemann's paper in these proceedings, which develops a type system that captures resource constraints that appear in compilers during register allocation.

4 Types at run-time

Many programming languages require compiled programs to manipulate some amount of type information at run-time. Interesting compilation issues arise when trying to make these run-time manipulations of types as efficient as possible. A prime example is the compilation of run-time type tests in dynamically-typed languages such as Scheme and Lisp: many clever tagging schemes have been developed to support fast run-time type tests. Another example is object-oriented languages such as Java, Modula-3, or C++ with run-time type inspection, where programs can dynamically test the actual class of an object. Again, clever encodings of the type hierarchy have been developed to perform those tests efficiently.

Even if the source language is fully statically typed, compilers and run-time systems may need to propagate type information to run-time in order to support certain operations. A typical example is the handling of non-parametric polymorphic operations such as polymorphic equality in ML and type classes in Haskell [41]. Another example is the handling of polymorphic records presented in [27]. There are several ways to precompile the required type information into an efficient form: one is to attach simple tags to data structures; another is to pass extra arguments (type representations or dictionaries of functions) to polymorphic functions. Elsman's paper in these proceedings compares the performances of these two approaches in the case of ML's polymorphic equality.

Passing run-time representations of type expressions as extra arguments to polymorphic function allows many type-directed compilation techniques to be applied to languages with polymorphic typing. The TIL compiler [39] and the Flint compiler [33] rely on run-time passing of type expressions (taken from extensions of the F_ω type system) to handle unboxed data structures in polymorphic functions and modules with abstract types. Constructing and passing these type expressions at run-time entail some execution overhead. The paper by Shao and Saha in these proceedings shows how to minimize this overhead by lifting those type-related computations out of loops and functions so that they all take place once at the beginning of program execution.

Non-conservative garbage collectors also require some amount of type information at run-time in order to distinguish pointers from non-pointers in memory roots and heap blocks. The traditional approach is to use tags on run-time values. Alternatively, Appel [4] suggested to attach source types to blocks of function code, and reconstruct type information for all reachable objects at run-time, using a variant of ML type reconstruction. The paper by Hosoya and Yonezawa in these proceedings is the first complete formalization of this approach.

Communicating static type information to the run-time system can be challenging, as it requires close cooperation from the compiler back-end. For instance, a type-directed garbage collector needs type information to be associated with registers and stack locations at garbage collection points; cooperation from the register allocator is needed to map the types of program variables onto the registers and stack slots. The paper by Bernard *et al.* in these proceedings discuss their experience with coercing a generic back-end into propagating type information.

Another operation that relies heavily on run-time type information is marshaling and un-marshaling between arbitrary data structures and streams of bytes – a crucial mechanism for persistence and distributed programming. In these proceedings, Duggan develops rich type systems to support marshaling in the presence of user-defined marshaling operations for some data types.

5 Typed intermediate languages

In traditional compiler technology, types are checked on the source language, but the intermediate representations used in the compilation process are essentially untyped. The intermediate representations maysometimes carry type annotations introduced by the front-end, but no provision is made for type-checking again these intermediate representations. Recently, several compilers have been developed that take the opposite approach: theirintermediate representations are equipped with typing rules and type-checking algorithms, and their various passes are presented as type-preserving transformations that, given a well-typed input, must produce a well-typed term of the target intermediate language.

The need for typed intermediate representations is obvious in compilers that require precise type information to be available till run-time, such as TIL and Flint [39, 33], or at least till late in the compilation process. Without requiring that each compiler pass be type-preserving and its output typable, it is nearly impossible to ensure the propagation of correct type information throughout the whole compiler.

Even in compilers that do not rely as crucially on types, typed intermediate languages can be extremely useful to facilitate the debugging of the compiler itself. During compiler development and testing, the type-checkers for the intermediate representations can be run on the outcome of every program transformation performed by the compiler. This catches a large majority of programming errors in the implementation of the transformations. In contrast with traditional compiler testing, which shows that the generated code is incorrect but does not indicate which pass is erroneous, type-checking the intermediate representations pinpoints precisely the culprit pass. The Glasgow Haskell compiler was one of the first to exploit systematically this technique [30].

So far, typed intermediate representations as described above have been applied almost exclusively to compiling functional languages. The paper by Wright *et al.* in these proceedings develop a typed intermediate language for compiling Java, and discuss the difficult issue of making explicit the "self" parameter to methods in a type-preserving way.

Typed intermediate languages usually do not go all the way down to code generation. For instance, Glasgow Haskell preserves types through its high-level program transformation, but the actual code generation is mostly untyped. The TIL compiler goes several steps further, in particular by performing the conversion of functions into closures in a type-preserving manner [24]. The paper by Morrisett *et al.* in these proceedings shows how to go all the way to assembly code: it proposes a type system for assembly code that can type-check reasonably optimized assembly code, including most uses of a stack.

6 Other applications of types

While the discussion above has concentrated on core compiler technology for functional and object-oriented languages, types have also found many exciting and sometimes unexpected applications in other areas of programming language implementation. For instance, type-directed partial evaluation is an interesting alternative to traditional partial evaluation based on source-level reductions. The paper by Balat and Danvy in these proceedings presents a type-directed partial evaluator that also uses run-time code generation. The paper by Fujinami presents a partial evaluator and run-time code generator for C++.

Languages for distributed programming based on process calculi are another area where the exploitation of type information is crucial to obtain good performances. Kobayashi's abstract in these proceedings surveys this topic.

Types have interesting applications in the area of language-based security for mobile code. Java applets have popularized the idea that foreign compiled code can be locally verified for type-correctness before execution. This local type-checking of compiled code then enables language-based security techniques that rely on typing invariants, such as the Java "sandbox". Advances in typed intermediate languages have an important impact in this area. For instance, while Java code verification is performed on unoptimized bytecode for an abstract machine, the paper by Morrisett *et al.* in these proceedings show that similar verifications can be carried on optimized machine code. Lee and Necula's work on proof-carrying code [26] show how to generalize this approach to the verification of arbitrary specifications.

In conclusion, there has been considerable cross-fertilization between type systems and compilers, and we hope to see more exciting new applications of types in the area of programming language implementations in the near future.

References

1. Ole Agesen, Jens Palsberg, and Michael Schwartzback. Type inference of Self: analysis of objects with dynamic and multiple inheritance. In *Proc. European*

Conference on Object-Oriented Programming – ECOOP'93, 1993.
2. Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools.* Addison-Wesley, 1986.
3. Alexander S. Aiken and Edward L. Wimmers. Type inclusion constraints and type inference. In *Functional Programming Languages and Computer Architecture 1993*, pages 31–41. ACM Press, 1993.
4. Andrew W. Appel. Run-time tags aren't necessary. *Lisp and Symbolic Computation*, 2(2), 1989.
5. David Bacon and Peter Sweeney. Fast static analysis of C++ virtual function calls. In *Object-Oriented Programming Systems, Languages and Applications '96*, pages 324–341. ACM Press, 1996.
6. Henry G. Baker. Unify and conquer (garbage, updating, aliasing, ...) in functional languages. In *Lisp and Functional Programming 1990*. ACM Press, 1990.
7. Anindya Banerjee. A modular, polyvariant, and type-based closure analysis. In *International Conference on Functional Programming 1997*, pages 1–10. ACM Press, 1997.
8. Lars Birkedal, Mads Tofte, and Magnus Vejlstrup. From region inference to von Neumann machines via region representation inference. In *23rd symposium Principles of Programming Languages*, pages 171–183. ACM Press, 1996.
9. Patrick Cousot. Types as abstract interpretations. In *24th symposium Principles of Programming Languages*, pages 316–331. ACM Press, 1997.
10. Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *Proc. European Conference on Object-Oriented Programming – ECOOP'95*, pages 77–101. Springer-Verlag, 1995.
11. Greg DeFouw, David Grove, and Craig Chambers. Fast interprocedural class analysis. In *25th symposium Principles of Programming Languages*, pages 222–236. ACM Press, 1998.
12. Allyn Dimock, Robert Muller, Franklyn Turbak, and J. B. Wells. Strongly typed flow-directed representation transformations. In *International Conference on Functional Programming 1997*, pages 11–24. ACM Press, 1997.
13. Amer Diwan, Kathryn S. McKinley, and J. Eliot B. Moss. Type-based alias analysis. In *Programming Language Design and Implementation 1998*, pages 106–117. ACM Press, 1998.
14. Jonathan Eifrig, Scott Smith, and Valery Trifonov. Type inference for recursively constrained types and its application to OOP. In *Mathematical Foundations of Programming Semantics*, volume 1 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1995.
15. Mary F. Fernández. Simple and effective link-time optimization of Modula-3 programs. In *Programming Language Design and Implementation 1995*, pages 103–115. ACM Press, 1995.
16. Robert Harper and Greg Morriset. Compiling polymorphism using intensional type analysis. In *22nd symposium Principles of Programming Languages* ACM Press, 1995.
17. Nevin Heintze. Set-based analysis of ML programs. In *Lisp and Functional Programming '94*, pages 306–317. ACM Press, 1994.
18. Fritz Henglein. Global tagging optimization by type inference. In *Lisp and Functional Programming 1992*. ACM Press, 1992.
19. Suresh Jagannathan and Andrew Wright. Polymorphic splitting: An effective polyvariant flow analysis. *ACM Transactions on Programming Languages and Systems*, 20(1):166–207, 1998.

20. Pierre Jouvelot and David K. Gifford. Algebraic reconstruction of types and effects. In *18th symposium Principles of Programming Languages*, pages 303–310. ACM Press, 1991.
21. Xavier Leroy. Unboxed objects and polymorphic typing. In *19th symposium Principles of Programming Languages*, pages 177–188. ACM Press, 1992.
22. Xavier Leroy. The effectiveness of type-based unboxing. In *Workshop Types in Compilation '97*. Technical report BCCS-97-03, Boston College, Computer Science Department, June 1997.
23. John M. Lucassen and David K. Gifford. Polymorphic effect systems. In *15th symposium Principles of Programming Languages*, pages 47–57. ACM Press, 1988.
24. Yasuhiko Minamide, Greg Morrisett, and Robert Harper. Typed closure conversion. In *23rd symposium Principles of Programming Languages*, pages 271–283. ACM Press, 1996.
25. John C. Mitchell. Coercion and type inference. In *11th symposium Principles of Programming Languages*, pages 175–185. ACM Press, 1984.
26. George C. Necula. Proof-carrying code. In *24th symposium Principles of Programming Languages*, pages 106–119. ACM Press, 1997.
27. Atsushi Ohori. A polymorphic record calculus. *ACM Transactions on Programming Languages and Systems*, 17(6):844–895, 1995.
28. Atsushi Ohori and Tomonobu Takamizawa. An unboxed operational semantics for ML polymorphism. *Lisp and Symbolic Computation*, 10(1):61–91, 1997.
29. Jens Palsberg and Patrick O'Keefe. A type system equivalent to flow analysis. In *22nd symposium Principles of Programming Languages*, pages 367–378. ACM Press, 1995.
30. Simon L. Peyton-Jones. Compiling Haskell by program transformation: a report from the trenches. In *European Symposium on Programming 1996*, volume 1058 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.
31. Simon L. Peyton-Jones and John Launchbury. Unboxed values as first-class citizens in a non-strict functional language. In *Functional Programming Languages and Computer Architecture 1991*, volume 523 of *Lecture Notes in Computer Science*, pages 636–666, 1991.
32. John Plevyak and Andrew Chien. Precise concrete type inference for object-oriented languages. In *Object-Oriented Programming Systems, Languages and Applications '94*, pages 324–340. ACM Press, 1994.
33. Zhong Shao. Flexible representation analysis. In *International Conference on Functional Programming 1997*, pages 85–98. ACM Press, 1997.
34. Zhong Shao and Andrew Appel. A type-based compiler for Standard ML. In *Programming Language Design and Implementation 1995*, pages 116–129. ACM Press, 1995.
35. Olin Shivers. Control-flow analysis in Scheme. In *Programming Language Design and Implementation 1988*, pages 164–174. ACM Press, 1988.
36. Olin Shivers. *Control-Flow Analysis of Higher-Order Languages.* PhD thesis, Carnegie Mellon University, May 1991.
37. Bjarne Steensgaard. Points-to analysis in almost linear time. In *23rd symposium Principles of Programming Languages*, pages 32–41. ACM Press, 1996.
38. Jean-Pierre Talpin and Pierre Jouvelot. The type and effect discipline. *Information and Computation*, 111(2):245–296, 1994.
39. D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: a type-directed optimizing compiler for ML. In *Programming Language Design and Implementation 1996*, pages 181–192. ACM Press, 1996.

40. Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.
41. Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad-hoc. In *16th symposium Principles of Programming Languages*, pages 60–76. ACM Press, 1989.