Where are you going with those types?

Vincent St-Amour, Sam Tobin-Hochstadt, Matthew Flatt, Matthias Felleisen

Northeastern University, University of Utah
{stamourv,samth,matthias}@ccs.neu.edu, mflatt@cs.utah.edu

Abstract. The use of unboxed data representations often increases the efficiency of programs, especially numerical ones. Operations on unboxed data do not need to mask out type tags or dereference pointers. As a result, certain operations, such as floating point arithmetic, become a single machine instruction.

Type-based techniques [1–3] directly enable the use of unboxed data representations in the presence of polymorphic functions. The problem is, however, that type information, computed in the front end, has to be carried though the entire compilation process, all the way to the code-generation phase, where unboxing decisions are made. Doing so increases the complexity of the compiler and results in a significant overhead in compilation time.

This work presents an alternative approach. Instead of pushing types through the compiler, it relies on exposing specialized primitives that operate on unboxed data of a given type. The typechecker then rewrites programs to use these primitives where it is safe to do so. That is, when there exists a specialized equivalent to the original primitive that can operate on values of the arguments' types, the type checker replaces the generic primitive with an equivalent that allows unboxing.

For example, Racket provides specialized arithmetic primitives that are only valid for floating-point numbers: `fl+`, `fl-` and so on. Typed Racket's typechecker replaces generic arithmetic primitives–regular Racket `+`, `-` and so on–with their specialized equivalents if it can prove that their arguments are floating-point numbers. Here is a sample rule from Typed Racket:

$$\texttt{(+ x y)} \rightarrow \texttt{(fl+ x y)} \quad \text{if } \Gamma \vdash x : \texttt{Float} \text{ and } \Gamma \vdash y : \texttt{Float}$$

We have implemented this approach as part of the Typed Racket [4, 5] system. Preliminary results show speedups comparable to the traditional typed-based approaches. We conjecture that this approach could be applied to other typed languages with generic operations such as Haskell and Standard ML.

References

1. Leroy, X.: Unboxed objects and polymorphic typing. In: ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. (1992) 177–188
2. Shao, Z., Appel, A.W.: A type-based compiler for Standard ML. In: ACM SIGPLAN Conference on Programming Language Design and Implementation. (1995) 116–129
3. Tarditi, D., Morrisett, G., Cheng, P., Stone, C., Harper, R., Lee, P.: TIL: A type-directed optimizing compiler for ML. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, ACM Press (1996) 181–192
4. Tobin-Hochstadt, S., Felleisen, M.: The design and implementation of Typed Scheme. In: ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ACM Press (2008) 395–406
5. Culpepper, R., Tobin-Hochstadt, S., Flatt, M.: Advanced macrology and the implementation of Typed Scheme. In: Workshop on Scheme and Functional Programming. (2007) 1–13