1 Unboxed objects and polymorphic typing

This paper presents a program transformation, representation analysis, that makes it possible for languages with polymorphic types, such as ML, to use unboxed data representations in monomorphic code. The solution the paper presents involves the use of coercions to convert back and forth between unboxed representations used by monomorphic functions to the boxed representation used by polymorphic functions. The paper then presents a formalization of the transformation and proofs of both its type correctness and semantic correctness. The technique is then extended to support concrete ML datatypes and mutable types. Finally an overview of the implementation of the transformation in the Gallium Caml Light compiler is presented, along with benchmarks comparing Gallium with other ML compilers.

1.2 Ideas

Fortran and Pascal compilers had been using type information to make compilation decisions for years. However even though ML, like Fortran and Pascal, is statically typed, the presence of parametric polymorphism reduces the representation techniques of Fortran compilers inapplicable. This paper presents ways to achieve some of these techniques, at least for monomorphic code. The result is that monomorphic code can have performance comparable to Fortran, while most of the code base still using polymorphism.
2 A type-based compiler for standard ML

This paper summarizes Leroy’s representation analysis technique and then goes on to explain how it is extended to work on full SML. Along the way, the paper explains in detail how representation analysis has been added to the SML/NJ compiler, and how it interacts with the rest of the system. It also explains some practical challenges and show solutions. It concludes by giving an evaluation of the benefits reaped from the addition of representation analysis to SML/NJ.

2.2 Ideas

This paper takes Leroy’s idea of representation analysis and extends it in various ways. First, they extend the technique to support the ML module system, and thus full SML. Second, they reduce the compilation time overhead due to type manipulation, which was a problem in Leroy’s work. They do so by using various optimizations and by simplifying the types for the back-end operate to the strict minimum necessary for coercions to work. Finally, they also increase the usefulness of representation analysis more useful by eliminating some amount of polymorphism from programs using minimum typing derivations, an idea due to Bjørner\(^1\).

\(^1\)see: Minimal Typing Derivations, by Nikolaj S. Bjørner (ML Workshop 1994)
3 TIL: A Type-Directed Optimizing Compiler for ML

@INPROCEEDINGS{til,
    author = {D. Tarditi and G. Morrisett and P. Cheng and C. Stone
              and R. Harper and P. Lee},
    title = {TIL: A Type-Directed Optimizing Compiler for ML},
    booktitle = {In ACM SIGPLAN Conference on Programming Language
                Design and Implementation},
    year = {1996},
    pages = {181--192},
    publisher = {ACM Press}
}

3.1 Content
This paper provides an overview of the design of the TIL compiler. It gives a
brief overview of intensional polymorphism and nearly tag-free garbage collec-
tion, the techniques that TIL relies on to achieve unboxing. The paper then
describes the structure of the TIL compiler, and how the above techniques fit
in. It then goes over a detailed example to explain the compilation process, and
concludes by evaluating the performance of the resulting system.

3.2 Ideas
Up to that point, work on using unboxed data representations in polymorphic
languages was limited to coercion-based approaches. These approaches have
downsides. They have to fall back on boxed representations in some cases,
and coercions can sometimes be costly. This paper introduces a new technique
to achieve unboxing without resorting to coercions. Their technique relies on
intensional polymorphism and nearly tag-free garbage collection. They keep
data unboxed at all times and construct, pass around and dispatch on type
information at runtime to compensate. As a result polymorphism becomes
expensive. The authors acknowledge this problem and take some first steps to
compensate.
4 Flexible Representation Analysis

@INPROCEEDINGS{flexible,
  author = {Zhong Shao},
  title = {Flexible Representation Analysis},
  booktitle = {In ACM SIGPLAN International Conference on Functional Programming},
  year = {1997},
  pages = {85--98},
  publisher = {ACM Press}
}

4.1 Content

The paper introduces flexible representation analysis, a descendent of representation analysis that supports a variety of boxed forms. The paper gives a brief overview of the existing techniques to achieve unboxing in polymorphic languages, and described the downsides of each. It then briefly explains flexible representation analysis and how it can fix these downsides. Afterwards, the paper formalizes the notion of canonical boxed forms and shows how to prove the validity of canonical boxed forms.

4.2 Ideas

This paper observes that representation analysis and intensional polymorphism are not the only approaches to achieve unboxing. In fact, the two are extremes on a continuum, and some representations in between would work as well. Thus, the amount of necessary boxing can be reduced and the amount of runtime type manipulation would need to be increased to compensate. Choice of canonical boxed representation becomes an engineering choice.
5 The Effectiveness of Type-Based Unboxing

The paper starts with an evaluation of the existing unboxing techniques used for polymorphic languages and identifies sources of overhead with all of them. These source of overhead are related to both polymorphic code performance and garbage collection, thus even monomorphic code is affected. The paper then presents some unboxing strategies that rely on little to no typing information and explains how they would compare to type-based approaches. Finally, benchmarks confirming their claims are provided.

5.2 Ideas

The previous approaches to unboxing all involved type information. They also had several drawbacks that could adversely affect their runtime performance, which were related to their use of types. Leroy presents some untyped or nearly untyped approaches to unboxing that achieve most of the benefits of type-based approaches, without their drawbacks. These approaches are therefore almost as good in the best case, and better in the worst case.