Abstract

The major weakness due to that functional languages such as ML become less popular than other imperative languages such as C is the inefficiency in senses of performance and memory usage. Developing a function language that can seamlessly interoperate with C (and other imperative languages) would help programmers to take advantage of both programming styles.

In this thesis, I present a type-directed compilation method of ML for achieving a high level of interoperability. This compilation method supports a memory management system, where integers, floating point values and other atomic data have the same natural representations as in other language implementations. This allows ML and other languages sharing the same heap space without any additional performance cost. Another advantage of this memory management model is that run-time system can have a better performance by eliminating all ``boxing" and ``tagging" operations appearing in most of conventional implementations.

In order to achieve this, I first consider an ``unboxed", ``non-tag" data representation model in which

- integers, floating point values and other atomic values are naturally represented;
- each a heap block or run-time environment (stack frame) has a ``bitmap" that describes the pointer positions in the block;

then develop the compilation method to support this model.

Since a polymorphic function may produce runtime objects of different types with different sizes, the compiler should be able to generate a function's code so that it has the same behavior for all instance types and it can compute a correct bitmap for each memory block.
This would require us to insert extra lambda abstractions and applications to pass the bit tags required in bitmap computation and the sizes required in manipulations on unboxed values.

This compilation process should be done for both stack frames and heap-allocated objects including function closures and their environment records. I solve the problem of mutual dependency between this compilation method and closure conversion by developing a combined algorithm that plays both of the roles. The resulting compilation process is shown to be sound with respect to an untyped operational semantics with bitmap-inspecting garbage collection.

I also consider several optimization techniques for reducing run-time overhead arising from bitmap computation and unboxed manipulation. This compilation method, together with the proposed optimizations, has been implemented in our SML# compiler for the full set of Standard ML language, demonstrating its practical feasibility.