In the Fall of 2005, Carnegie Mellon offered for the first time a course on compilers for higher-order, typed programming languages designed to be accessible for students at both the graduate and undergraduate levels. In its first iteration, HOT Compilation's projects explored the compilation of Standard ML from elaboration to CPS conversion. With a few notable exceptions, the material in the course was presented very much like "From System F to Typed Assembly Language" [3], which explicated the translation from a traditional typed lambda calculus to calculi that were each successively closer to real machine code. As part of the independent project component of that course, the members of this group implemented subsequent phases of the translation down to TAL, such as closure conversion and hoisting of closed functions. However, due to time constraints no student participating in that course completed a full compiler down to executable code.

We propose implementing a type-preserving back-end for a HOT Compilation compiler, going from closure converted code to an idealized typed assembly language we will design for the purposes of this project. Following traditional compiler design principles, we will translate our lowest intermediate language to a version of our typed assembly language with an infinite number of registers. Instruction selection will be a fairly trivial issue, as we intend to have a RISC-like syntax for our assembly language which will align cleanly with most operations extant in the lower intermediate languages. From this point translating the code to a machine with finite registers requires performing register allocation. To do this, we intend to implement a Pereira-Palsberg style register allocator [4], which is known to be optimal for chordal interference graphs. As our lower intermediate languages are already CPS converted, they are essentially already in SSA form and thus possess chordal interference graphs. An interesting consequence of starting from a CPS converted/closure converted source language is that we will not have have ϕ-nodes. The non-existence of ϕ-nodes obviates the need to perform SSA elimination after code generation. However, this simplification is paid for by the cost of creating and passing around the environments that were introduced during closure conversion. These environments effectively act as ϕ-nodes.

In order to test our back-end, we will also write a simulator for our typed assembly language. Designing our own typed assembly language will allow us to specialize the language for the register allocation problem, rather than employing a much more heavyweight system designed to capture a foundational typed assembly language [1] or to add a type system to an existing commonly used architecture [2]. This allows us to simplify issues that complicate more realistic TALs, such as initializing tuples in a type-safe way and representing sum types at the machine level. In order to be faithful to the register allocation problem, our TAL will have finite registers, a stack, a heap, and some notion of calling conventions. We hope to make a TAL that is parameterized over different calling conventions and the number of registers, in order to obtain empirical data about the performance of our register allocator over different instantiations of these parameters. The empirical data we collect will allow us to infer statements about the register and memory demands of fairly naively generated code for a higher-order functional programming language. If time permits, we will also attempt to record the performance benefits of a number of simple but currently undetermined optimizations. As we will be writing our own simulator for the language, we have the opportunity to obtain very detailed information about the runtime behavior of our generated code without having to instrument our output.

The original Pereira-Palsberg register allocator was implemented for a Java compiler, where only some high percentage of interference graphs were chordal. In contrast we expect all interference graphs to be chordal for our source language. As this result is fairly new, we do not know of any existing ML compilers which employ this particular algorithm. This same technique could feasibly be attempted on a more mature ML compiler or target a more sophisticated TAL. However, we have chosen a course project compiler and an idealized TAL in order to ensure a clean and speedy implementation. Given the nature of this project, we will not require resources more complicated than access to machines which have SML/NJ installed. Although concessions have been made to increase the likelihood of successful completion within the time given, we feel the goals of this project still capture the problem of code generation and register allocation for a type preserving compiler. In addition, we feel the experience we gain and the artifacts we produce may have pedagogical value for future students studying code generation for modern programming languages.

References

[1] Karl Crary. Toward a foundational typed assembly language. In *Symposium on Principles of Programming Languages*, pages 198–212, 2003.

[2] Greg Morrisett, Karl Crary, Neal Glew, Dan Grossman, Richard Samuels, Frederick Smith, Daivd Walker, Stephanie Weirich, and Steve Zdancewic. TALx86: A realistic type assembly language. In *Workshop on Compiler Support for System Software*, pages 25–35, Atlanta, GA, USA, May 1999. INRIA Research Report 0228.

[3] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From system f to typed assembly language. *ACM Transactions on Programming Languages and Systems.*, 21(3):527–568, 1999.

[4] Fernando Magno Quintão Pereira and Jens Palsberg. Register allocation via coloring of chordal graphs. In Kwangkeun Yi, editor, *APLAS*, volume 3780 of *Lecture Notes in Computer Science*, pages 315–329. Springer, 2005.