If programs are instructions for computers, then types describe the *intent* of those instructions. The *type system* of the programs' language characterizes the kind of intentions we can express. A simple type system might only allow expressing simple intentions, such as anticipating a float and not an integer at a given location, or expecting that a particular function take two integers and return one.

More complex type systems, such as *dependent type systems*, allow us to express more complex intentions. For instance, the type of an array could assert that the array has some particular length, and the type of a function that indexes into the array could enforce that the index is no greater than the length of the array. This can be done since the types themselves can contain programs. For the index bounds check in the type, we can use $\leq$, which itself would be a recursive function that takes two naturals. We can then skip this check in the function body, as the type assures us that it functions as we intend.

Or so we think. Generally, most programmers don't write directly in assembly, and require that a compiler translate their programs from a high-level language. Often the process of compiling a program simplifies or erases more complex types, which can cause problems. Consider the following scenarios:

1. If types are erased, then we cannot automatically check that our compiled indexing function never tries to index beyond the length of the array. In other words, there is the possibility of a *miscompilation error*, where our function behaves differently from how we've specified it in the type.

2. A compiled program rarely exists in a vacuum; it may link to (that is, call on) other programs, and other programs may link to it. If types are erased, then we cannot guarantee that our compiled indexing function is always called with an index no greater than the length of the array. In other words, there is the possibility of a *linking error*, where the invariants that we assume to hold in the function body are violated.

The most straightforward solution would be to *compile the types too* (Morrisett, Walker, Crary, & Glew, 1999). We eliminate miscompilation errors by showing that compiled programs always correspond to their compiled types, since this represents the intent of the program being preserved. Similarly, we avoid linking errors by type checking the compiled types of the two compiled programs before linking.

The difficulty lies in the need for a dependent type system to express even our simple guaranteed indexing function, as the complexity of these systems makes correct compilation complicated. Since we need type checking to terminate, we need the programs in the types to terminate as well. In particular, as we've seen with the index bounds check example, we need all recursive functions to terminate.

The conventional method of checking the termination of recursive functions is to inspect them syntactically (Giménez, 1994). They are provably terminating if they satisfy certain syntactic conditions ensuring that recursive calls occur only on smaller and smaller arguments. Unfortunately, as programs are compiled closer and closer to assembly, they get broken up into more and more pieces; checking the syntactic conditions amounts to keeping track of all the program pieces, which quickly becomes infeasible.

Alternatively, we can check termination by adding *size* information to the types (Hughes, Pareto, & Sabry, 1996). Recursive functions will then terminate if recursive calls occur on arguments whose types have smaller and smaller sizes; termination checking then becomes type checking. This technique is more robust than relying merely on syntactic checks, at the expense of even more complex types.

While there is existing work on compiling dependently-typed programs and their types (Bowman, 2018) and on adding sizes to dependent types (Barthe, Grégoire, & Pastawski, 2006), there is none on compiling dependently-typed programs *and their sized types*. **In my thesis, I will be exploring the feasibility of compiling sized dependent types as an alternative to syntactic checks for termination checking.** In particular, I will be tackling the compiler pass of *closure conversion*, which turns functions that can be nested within each other into independent *closures*. This pass is particularly interesting because the recursiveness of recursive functions needs to be reflected in some way in the closures that they compile to, which is tricky for dependent types.

References

Barthe, G., Grégoire, B., & Pastawski, F. (2006). CIC^: Type-Based Termination of Recursive Definitions in the Calculus of Inductive Constructions. In M. Hermann, & A. Voronkov (Ed.), *Logic for Programming, Artificial Intelligence, and Reasoning. 3461*, pp. 71-85. Berlin/Heidelberg: Springer-Verlag. doi:10.1007/11916277_18

Bowman, W. J. (2018). *Compiling with Dependent Types.* Northeastern University, Computer Science. Boston: Northeastern University. Retrieved from http://hdl.handle.net/2047/D20316239

Giménez, E. (1994). Codifying guarded definitions with recursive schemes. In P. Dybjer, B. Nordström, & J. Smith (Ed.), *Types for Proofs and Programs. 996*, pp. 39-59. Berlin/Heidelberg: Springer-Verlag. doi:10.1007/3-540-60579-7_3

Hughes, J., Pareto, L., & Sabry, A. (1996). Proving the Correctness of Reactive Systems using Sized Types. *Principles of Programming Languages* (pp. 410-423). New York: Association for Computing Machinery. doi:10.1145/237721

Morrisett, G., Walker, D., Crary, K., & Glew, N. (1999, May 1). From System F to Typed Assembly Language. *ACM Transactions on Programming Languages and Systems, 21*(3), 85-97. doi:10.1145/319301