A Certified Type-Preserving Compiler from Lambda Calculus to Assembly Language *

Adam Chlipala
University of California, Berkeley
adamc@cs.berkeley.edu

Abstract

We present a certified compiler from the simply-typed lambda calculus to assembly language. The compiler is certified in the sense that it comes with a machine-checked proof of semantics preservation, performed with the Coq proof assistant. The compiler and the terms of its several intermediate languages are given dependent types that guarantee that only well-typed programs are representable. Thus, type preservation for each compiler pass follows without any significant "proofs" of the usual kind. Semantics preservation is proved based on denotational semantics assigned to the intermediate languages. We demonstrate how working with a type-preserving compiler enables type-directed proof search to discharge large parts of our proof obligations automatically.

Categories and Subject Descriptors F.3.1 [*Logics and meanings of programs*]: Mechanical verification; D.2.4 [*Software Engineering*]: Correctness proofs, formal methods, reliability; D.3.4 [*Programming Languages*]: Compilers

General Terms Languages, Verification

Keywords compiler verification, interactive proof assistants, dependent types, denotational semantics

1. Introduction

Compilers are some of the most complicated pieces of widely-used software. This fact has unfortunate consequences, since almost all of the guarantees we would like our programs to satisfy depend on the proper workings of our compilers. Therefore, proofs of compiler correctness are at the forefront of useful formal methods research, as results here stand to benefit many users. Thus, it is not surprising that recently and historically there has been much interest in this class of problems. This paper is a report on our foray into that area and the novel techniques that we developed in the process.

One interesting compiler paradigm is *type-directed compilation*, embodied in, for instance, the TIL Standard ML compiler [TMC$^+$96]. Where traditional compilers use relatively type-impoverished intermediate languages, TIL employed *typed intermediate languages* such that every intermediate program had a typing derivation witnessing its safety, up until the last few phases of compilation. This type information can be used to drive important optimizations, including *nearly tag-free garbage collection*, where the final binary comes with a table giving the type of each register or stack slot at each program point where the garbage collector may be called. As a result, there is no need to use any dynamic typing scheme for values in registers, such as tag bits or boxing.

Most of the intricacies of TIL stemmed from runtime passing of type information to support polymorphism. In the work we present here, we instead pick the modest starting point of simply-typed lambda calculus, but with the same goal: we want to compile the programs of this calculus into an idealized assembly language that uses nearly tag-free garbage collection. We will achieve this result by a series of six type-directed translations, with their typed target languages maintaining coarser and coarser grained types as we proceed. Most importantly, we prove *semantics preservation* for each translation and compose these results into a machine-checked correctness proof for the compiler. To our knowledge, there exists no other computer formalization of such a proof for a type-preserving compiler.

At this point, the reader may be dubious about just how involved a study of simply-typed lambda calculus can be. We hope that the exposition in the remainder of this paper will justify the interest of the domain. Perhaps the key difficulty in our undertaking has been effective handling of variable binding. The POPLmark Challenge [ABF$^+$05] is a benchmark initiative for computer formalization of programming languages whose results have highlighted just that issue. In the many discussions it has generated, there has yet to emerge a clear winner among basic techniques for representing language constructs that bind variables in local scopes. Each of the proposed techniques leads to some significant amount of overhead not present in traditional proofs. Moreover, the initial benchmark problem doesn't involve any component of relational reasoning, crucial for compiler correctness. We began this work as an investigation into POPLmark-style issues in the richer domain of relational reasoning.

1.1 Task Description

The source language of our compiler is the familiar simply-typed lambda calculus, whose syntax is:

$$\begin{array}{rrcl} \text{Types} & \tau & ::= & \mathsf{N} \mid \tau \to \tau \\ \\ \text{Natural numbers} & n & & \\ \text{Variables} & x, y, z & & \\ \text{Terms} & e & ::= & n \mid x \mid e\ e \mid \lambda x : \tau.\ e \end{array}$$

Application, the third production for e, associates to the left, as usual. We will elaborate shortly on the language's semantics.

* This research was supported in part by a National Defense Science and Engineering Graduate Fellowship, as well as National Science Foundation grants CCF-0524784 and CCR-0326577.

PLDI'07 June 11–13, 2007, San Diego, California, USA.

Our target language is an idealized assembly language, with infinitely many registers and memory cells, each of which holds unbounded natural numbers. We model interaction with a runtime system through special instructions. The syntax of the target language is:

$$\begin{array}{rrcl} \text{Registers} & r & & \\ \text{Operands} & o & ::= & r \mid n \mid \mathsf{new}\,\langle \vec{r}, \vec{r} \rangle \mid \mathsf{read}\,\langle r, n \rangle \\ \text{Instructions} & i & ::= & r := o; i \mid \mathsf{jump}\ r \\ \text{Programs} & p & ::= & \langle \vec{i}, i \rangle \end{array}$$

As instruction operands, we have registers, constant naturals, and allocation and reading of heap-allocated records. The new operand takes two arguments: a list of root registers and a list of registers holding the field values for the object to allocate. The semantics of the target language are such that new is allowed to perform garbage collections at will, and the root list is the usual parameter required for sound garbage collection. In fact, we will let new rearrange memory however it pleases, as long as the result is indistinguishable from having simply allocated a new record, from the point of view of the specified roots. The read instruction determines which runtime system-specific procedure to use to read a given field of a record.

A program is a list of basic blocks plus an initial basic block, where execution begins. A basic block is a sequence of register assignments followed by an indirect jump. The basic blocks are indexed in order by the natural numbers for the purposes of these jumps. We will additionally consider that a jump to the value 0 indicates that the program should halt, and we distinguish one register that is said to hold the program's result at such points.

We are now ready to state informally the theorem whose proof is the goal of this work:

THEOREM 1 (Informal statement of compiler correctness). *Given a term e of the simply-typed lambda calculus of type* N*, the compilation of e is an assembly program that, when run, terminates with the same result as we obtain by running e.*

Our compiler itself is implemented entirely within the Coq proof assistant [BC04]. Coq's logic doubles as a functional programming language. Through the *program extraction* process, the more exotic features of a development in this logic can be erased in a semantics-preserving way, leaving an OCaml program that can be compiled to efficient object code. In this way, we obtain a traditional executable version of our compiler. We ignore issues of parsing in this work, so our compiler must be composed with a parser. Assuming a fictitious machine that runs our idealized assembly language, the only remaining piece to be added is a pretty-printer from our abstract syntax tree representation of assembly programs to whatever format that machine requires.

1.2 Contributions

We summarize the contributions of this work as follows:

- It includes what is to our knowledge the first total correctness proof of an entire type-preserving compiler.
- It gives the proof using denotational semantics, in contrast to the typical use of operational semantics in related work.
- The whole formal development is carried out in a completely rigorous way with the Coq proof assistant [BC04], yielding a proof that can be checked by a machine using a relatively small trusted code base. Our approach is based on a general methodology for representing variable binding and denotation functions in Coq.
- We sketch a generic programming system that we have developed for automating construction of syntactic helper functions over de Bruijn terms [dB72], as well as generic correctness proofs about these functions. In addition, we present the catalogue of generic functions that we found sufficient for this work.
- Finally, we add to the list of pleasant consequences of making your compiler type-directed or type-preserving. In particular, we show how dependently-typed formalizations of type-preserving compilers admit particularly effective automated proof methods, driven by type information.

1.3 Outline

In the next section, we present our compiler's intermediate languages and elaborate on the semantics of the source and target languages. Following that, we run through the basic elements of implementing a compiler pass, noting challenges that arise in the process. Each of the next sections addresses one of these challenges and our solution to it. We discuss how to mechanize typed programming languages and transformations over them, how to use generic programming to simplify programming with dependently-typed abstract syntax trees, and how to apply automated theorem proving to broad classes of proof obligations arising in certification of type-preserving compilers. After this broad discussion, we spend some time discussing interesting features of particular parts of our formalization. Finally, we provide some statistics on our implementation, discuss related work, and conclude.

2. The Languages

In this section, we present our source, intermediate, and target languages, along with their static and dynamic semantics. Our language progression is reminiscent of that from the paper "From System F to Typed Assembly Language" [MWCG99]. The main differences stem from the fact that we are interested in meaning preservation, not just type safety. This distinction makes our task both harder and easier. Naturally, semantics preservation proofs are more difficult than type preservation proofs. However, the fact that we construct such a detailed understanding of program behavior allows us to retain less type information in later stages of the compiler.

The mechanics of formalizing these languages in Coq is the subject of later sections. We will stick to a "pencil and paper formalization" level of detail in this section. We try to use standard notations wherever possible. The reader can rest assured that anything that seems ambiguous in the presentation here is clarified in the mechanization.

2.1 Source

The syntax of our source language (called "Source" hereafter) was already given in Section 1.1. The type system is the standard one for simply-typed lambda calculus, with judgments of the form $\Gamma \vdash e : \tau$ which mean that, with respect to the type assignment to free variables provided by Γ, term e has type τ.

Following usual conventions, we require that, for any typing judgment that may be stated, let alone verified to hold, all free and bound variables are distinct, and all free variables of the term appear in the context. In the implementation, the first condition is discharged by using de Bruijn indices, and the second condition is covered by the dependent typing rules we will assign to terms.

Next we need to give Source a dynamic semantics. For this and all our other languages, we opted to use denotational semantics. The utility of this choice for compiler correctness proofs has been understood for a while, as reifying a program phrase's meaning as a mathematical object makes it easy to transplant that meaning to new contexts in modeling code transformations.

Our semantics for Source is:

$$\begin{array}{rcl} [\![\tau]\!] & : & \text{types} \rightarrow \text{sets} \\ [\![\mathsf{N}]\!] & = & \mathbb{N} \\ [\![\tau_1 \rightarrow \tau_2]\!] & = & [\![\tau_1]\!] \rightarrow [\![\tau_2]\!] \end{array}$$

$$\begin{array}{rcl} [\![\Gamma]\!] & : & \text{contexts} \rightarrow \text{sets} \\ [\![\cdot]\!] & = & \mathsf{unit} \\ [\![\Gamma, x : \tau]\!] & = & [\![\Gamma]\!] \times [\![\tau]\!] \end{array}$$

$$\begin{array}{rcl} [\![e]\!] & : & [\Gamma \vdash e : \tau] \rightarrow [\![\Gamma]\!] \rightarrow [\![\tau]\!] \\ [\![n]\!]\sigma & = & \overline{n} \\ [\![x]\!]\sigma & = & \sigma(x) \\ [\![e_1\ e_2]\!]\sigma & = & [\![e_1]\!]\sigma\ [\![e_2]\!]\sigma \\ [\![\lambda x : \tau.\ e]\!]\sigma & = & \lambda x : [\![\tau]\!].\ [\![e]\!](\sigma, x) \end{array}$$

The type of the term denotation function (used in expressions of the form $[\![e]\!]$) indicates that it is a function whose domain is *typing derivations* for terms and whose range depends on the particular Γ and τ that appear in that derivation. As a result, we define denotations only for well-typed terms.

Every syntactic class is being compiled into a single meta language, Coq's Calculus of Inductive Constructions (CIC) [BC04]. However, we will not expect any special knowledge of this formal system from the reader. Various other type theories could be substituted for CIC, and standard set theory would do just as well for this informal presentation.

We first give each Source type a meaning by a recursive translation into sets. Note that, throughout this paper, we overload constructs like the function type constructor $\rightarrow$ that are found in both our object languages and the meta language CIC, in an effort to save the reader from a deluge of different arrows. The variety of arrow in question should always be clear from context. With this convention in mind, the type translation for Source is entirely unsurprising. $\mathbb{N}$ denotes the mathematical set of natural numbers, as usual.

We give a particular type theoretical interpretation of typing contexts as tuples, and we then interpret a term that has type τ in context Γ as a function from the denotation of Γ into the denotation of τ. The translation here is again entirely standard. For the sake of conciseness, we allow ourselves to be a little sloppy with notations like $\sigma(x)$, which denotes the proper projection from the tuple σ, corresponding to the position x occupies in Γ. We note that we use the *meta language's lambda binder* to encode the object language's lambda binder in a natural way, in an example closely related to higher-order abstract syntax [PE88].

2.2 Linear

Our first compilation step is to convert Source programs into a form that makes *execution order* explicit. This kind of translation is associated with continuation-passing style (CPS), and the composition of the first two translations accomplishes a transformation to CPS. The result of the first translation is the Linear language, and we now give its syntax. It inherits the type language of Source, though we interpret the types differently.

$$\begin{array}{rrcl} \text{Operands} & o & ::= & n \mid x \mid \lambda x : \tau.\ e \\ \text{Terms} & e & ::= & \mathsf{let}\ x = o\ \mathsf{in}\ e \mid \mathsf{throw}\ \langle o \rangle \mid x\ y\ z \end{array}$$

Linear terms are linearized in the sense that they are sequences of binders of primitive operands to variables, followed by either a "throw to the current continuation" or a function call. The function call form shows that functions take two arguments: first, the normal argument; and second, a function to call with the result. The function and both its arguments must be variables, perhaps bound with earlier lets.

For reasons of space, we omit the standard typing rules for Linear and proceed to its denotational semantics. Recall, however, that the denotation functions are only defined over well-typed terms.

$$\begin{array}{rcl} [\![\tau]\!] & : & \text{types} \rightarrow \text{sets} \\ [\![\mathsf{N}]\!] & = & \mathbb{N} \\ [\![\tau_1 \rightarrow \tau_2]\!] & = & [\![\tau_1]\!] \rightarrow ([\![\tau_2]\!] \rightarrow \mathbb{N}) \rightarrow \mathbb{N} \end{array}$$

$$\begin{array}{rcl} [\![o]\!] & : & [\Gamma \vdash o : \tau] \rightarrow [\![\Gamma]\!] \rightarrow [\![\tau]\!] \\ [\![n]\!]\sigma & = & \overline{n} \\ [\![x]\!]\sigma & = & \sigma(x) \\ [\![\lambda x : \tau.\ e]\!]\sigma & = & \lambda x : [\![\tau]\!].\ [\![e]\!](\sigma, x) \end{array}$$

$$\begin{array}{rcl} [\![e]\!] & : & [\Gamma \vdash o : \tau] \rightarrow [\![\Gamma]\!] \rightarrow ([\![\tau]\!] \rightarrow \mathbb{N}) \rightarrow \mathbb{N} \\ [\![\mathsf{let}\ x = o\ \mathsf{in}\ e]\!]\sigma & = & [\![e]\!](\sigma, [\![o]\!]\sigma) \\ [\![\mathsf{throw}\ \langle o \rangle]\!]\sigma & = & \lambda k.\ k([\![o]\!]\sigma) \\ [\![x\ y\ z]\!]\sigma & = & \lambda k.\ \sigma(x)\ (\sigma(y))\ (\lambda v.\ k(\sigma(z)(v))) \end{array}$$

We choose the natural numbers as the result type of programs. Functions are interpreted as accepting continuations that return naturals when given a value of the range type. The let case relies on the implicit fact that the newly-bound variable x falls at the end of the proper typing context for the body e. The most interesting case is the last one listed, that for function calls. We call the provided function with a new continuation created by composing the current continuation with the continuation supplied through the variable z.

We use Linear as our first intermediate language instead of going directly to standard CPS because the presence of distinguished throw terms makes it easier to optimize term representation by splicing terms together. This separation is related to the issue of "administrative redexes" in standard two-phase CPS transforms [Plo75].

2.3 CPS

The next transformation finishes the job of translating Linear into genuine CPS form, and we call this next target language CPS.

$$\begin{array}{rrcl} \text{Types} & \tau & ::= & \mathsf{Nat} \mid \vec{\tau} \rightarrow \mathbb{N} \\ \text{Operands} & o & ::= & n \mid x \mid \lambda \vec{x} : \vec{\tau}.\ e \\ \text{Terms} & e & ::= & \mathsf{let}\ x = o\ \mathsf{in}\ e \mid x\ \vec{y} \end{array}$$

Compared to Linear, the main differences we see are that functions may now take multiple arguments and that we have collapsed throw and function call into a single construct. In our intended use of CPS, functions will either correspond to source-level functions and take two arguments, or they will correspond to continuations and take single arguments. Our type language has changed to reflect that functions no longer return, but rather they lead to final natural number results.

We omit discussion of the semantics of CPS, since the changes from Linear are quite incremental.

2.4 CC

The next thing the compiler does is closure convert CPS programs, hoisting all function definitions to the top level and changing those functions to take records of their free variables as additional arguments. We call the result language CC, and here is its syntax.

$$\begin{array}{rrcl} \text{Types} & \tau & ::= & \mathsf{N} \mid \vec{\tau} \rightarrow \mathbb{N} \mid \bigotimes \vec{\tau} \mid \vec{\tau} \times \vec{\tau} \rightarrow \mathbb{N} \\ \text{Operands} & o & ::= & n \mid x \mid \langle x, \vec{y} \rangle \mid \pi_i x \\ \text{Terms} & e & ::= & \mathsf{let}\ x = o\ \mathsf{in}\ e \mid x\ \vec{y} \\ \text{Programs} & p & ::= & \mathsf{let}\ x = (\lambda \vec{y} : \vec{\tau}.\ e)\ \mathsf{in}\ p \mid e \end{array}$$

We have two new type constructors: $\bigotimes \vec{\tau}$ is the multiple-argument product type of records whose fields have the types given by $\vec{\tau}$. $\vec{\tau} \times \vec{\tau} \rightarrow \mathbb{N}$ is the type of *code pointers*, specifying first the type of the closure environment expected and second the types of the regular arguments.

Among the operands, we no longer have an anonymous function form. Instead, we have $\langle x, \vec{y} \rangle$, which indicates the creation of a closure with code pointer x and environment elements $\vec{y}$. These environment elements are packaged atomically into a record, and the receiving function accesses the record's elements with projection operand form $\pi_i x$. This operand denotes the ith element of record x.

While we have added a number of features, there are no surprises encountered in adapting the previous semantics, so we will proceed to the next language.

2.5 Alloc

The next step in compilation is to make allocation of products and closures explicit. Since giving denotational semantics to higher-order imperative programs is tricky, we decided to perform "code flattening" as part of the same transformation. That is, we move to first-order programs with fixed sets of numbered code blocks, and every function call refers to one of these blocks by number. Here is the syntax of this language, called Alloc.

$$
\begin{array}{rrcl}
\text{Types} & \tau & ::= & \mathsf{N} \mid \mathsf{ref} \mid \vec{\tau} \rightarrow \mathbb{N} \\
\text{Operands} & o & ::= & n \mid x \mid \langle n \rangle \mid \mathsf{new}\ \langle \vec{x} \rangle \mid \pi_i x \\
\text{Terms} & e & ::= & \mathsf{let}\ x = o\ \mathsf{in}\ e \mid x\ \vec{y} \\
\text{Programs} & p & ::= & \mathsf{let}\ (\lambda \vec{y} : \vec{\tau}.\ e)\ \mathsf{in}\ p \mid e
\end{array}
$$

The first thing to note is that this phase is the first in which we lose type information: we have a single type ref for all heap references, forgetting what we know about record field types. To compensate for this loss of information, we will give Alloc programs a semantics that allows them to fail when they make "incorrect guesses" about the types of heap cells.

Our set of operands now includes both natural number constants n and code pointer constants $\langle n \rangle$. We also have a generic record allocation construct in place of the old closure constructor, and we retain projections from records.

This language is the first to take a significant departure from its predecessors so far as dynamic semantics is concerned. The key difference is that Alloc admits non-terminating programs. While variable scoping prevented cycles of function calls in CC and earlier, we lose that restriction with the move to a first-order form. This kind of transformation is inevitable at some point, since our target assembly language has the same property.

Domain theory provides one answer to the questions that arise in modeling non-terminating programs denotationally, but it is difficult to use the classical work on domain theory in the setting of constructive type theory. One very effective alternative comes in the form of the *co-inductive types* [Gim95] that Coq supports. A general knowledge of this class of types is not needed to understand what follows. We will confine our attention to one co-inductive type, a sort of *possibly-infinite streams*. We define this type with the infinite closure of this grammar:

$$
\text{Traces} \quad T \quad ::= \quad n \mid \bot \mid \star, T
$$

In other words, a *trace* is either an infinite sequence of stars or a finite sequence of stars followed by a natural number or a bottom value. The first of these possibilities will be the denotation of a non-terminating program, the second will denote a program returning an answer, and the third will denote a program that "crashes."

Now we are ready to start modeling the semantics of Alloc. First, we need to fix a representation of the heap. We chose an abstract representation that identifies heaps with lists of lists of tagged fields, standing for the set of finite-size records currently allocated. Each field consists of a tag, telling whether or not it is a pointer; and a data component. As we move to lower languages, these abstract heaps will be mapped into more conventional flat, untyped heaps. We now define some domains to represent heaps, along with a function for determining the proper tag for a type:

$$
\begin{array}{rcl}
\mathbb{C} & = & \{\mathsf{Traced}, \mathsf{Untraced}\} \times \mathbb{N} \\
\mathbb{M} & = & \mathsf{list}\ (\mathsf{list}\ \mathbb{C}) \\
\mathsf{tagof}(\mathsf{N}) & = & \mathsf{Untraced} \\
\mathsf{tagof}(\mathsf{ref}) & = & \mathsf{Traced} \\
\mathsf{tagof}(\vec{\tau} \rightarrow \mathbb{N}) & = & \mathsf{Untraced}
\end{array}
$$

Next we give a semantics to operands. We make two different choices here than we did before. First, we allow execution of operands to *fail* when a projection reads a value from the heap and finds that it has the wrong tag. Second, the denotations of operands must be heap transformers, taking the heap as an extra argument and returning a new one to reflect any changes. We also set the convention that program counter 0 denotes the distinguished top-level continuation of the program that, when called, halts the program with its first argument as the result. Any other program counter $n+1$ denotes the nth function defined in the program.

$$
\begin{array}{rcl}
[\![o]\!] & : & [\Gamma \vdash o : \tau] \rightarrow [\![\Gamma]\!] \rightarrow \mathbb{M} \rightarrow (\mathbb{M} \times \mathbb{N}) \cup \{\bot\} \\
[\![n]\!]\sigma m & = & (m, \overline{n}) \\
[\![x]\!]\sigma m & = & (m, \sigma(x)) \\
[\![\langle n \rangle]\!]\sigma m & = & (m, \overline{n+1}) \\
[\![\mathsf{new}\ \langle \vec{x} \rangle]\!]\sigma m & = & (m \oplus [\sigma(\vec{x})], |m|) \\
[\![\pi_i x]\!]\sigma m & = & \textbf{if}\ m_{\sigma(x),i} = (\mathsf{tagof}(\tau), v),\ \textbf{then}\text{: } (m, v) \\
 & & \textbf{else}\text{: } \bot
\end{array}
$$

We write $\oplus$ to indicate list concatenation, and the notation $\sigma(\vec{x})$ to denote looking up in σ the value of each variable in $\vec{x}$, forming a list of results where each is tagged appropriately based on the type of the source variable. The new case returns the length of the heap because we represent heap record addresses with their zero-based positions in the heap list.

Terms are more complicated. While one might think of terms as potentially non-terminating, we take a different approach here. The denotation of every term is a terminating program that returns *the next function call that should be made*, or signals an error with a $\bot$ value. More precisely, a successful term execution returns a tuple of a new heap, the program counter of the function to call, and a list of natural numbers that are to be the actual arguments.

$$
\begin{array}{rcl}
[\![e]\!] & : & [\Gamma \vdash e : \tau] \rightarrow [\![\Gamma]\!] \rightarrow \mathbb{M} \\
 & & \rightarrow (\mathbb{M} \times \mathbb{N} \times \mathsf{list}\ \mathbb{N}) \cup \{\bot\} \\
[\![\mathsf{let}\ x = o\ \mathsf{in}\ e]\!]\sigma m & = & \textbf{if}\ [\![o]\!]\sigma m = (m', v)\ \textbf{then}\text{: } [\![e]\!](\sigma, v)m' \\
 & & \textbf{else}\text{: } \bot \\
[\![x\ \vec{y}]\!]\sigma m & = & (m, \sigma(x), \sigma(\vec{y}))
\end{array}
$$

Finally, we come to the programs, where we put our trace domain to use. Since we have already converted to CPS, there is no need to consider any aspect of a program's behavior but its result. Therefore, we interpret programs as functions from heaps to traces. We write :: for the binary operator that concatenates a new element to the head of a list, and we write $\vec{x}_{pc}$ for the lookup operation that extracts from the function definition list $\vec{x}$ the pcth element.

$$\begin{array}{rcl}
[\![p]\!] & : & [\Gamma \vdash \mathsf{let}\ \vec{x}\ \mathsf{in}\ e] \to [\![\Gamma]\!] \to \mathbb{M} \to \mathsf{Trace} \\
[\![\mathsf{let}\ \vec{x}\ \mathsf{in}\ e]\!]\sigma m & = & \textbf{if}\ [\![e]\!]\sigma m = (m', 0, v :: \vec{n})\ \textbf{then}\text{: } v \\
& & \textbf{else if}\ [\![e]\!]\sigma m = (m', pc+1, \vec{n})\ \textbf{then}\text{:} \\
& & \quad \textbf{if}\ \vec{x}_{pc} = \lambda \vec{y} : \vec{\tau}.\ e'\ \textbf{and}\ |\vec{y}| = |\vec{n}|\ \textbf{then}\text{:} \\
& & \quad\quad \star, [\![\mathsf{let}\ \vec{x}\ \mathsf{in}\ e']\!]\vec{n}m' \\
& & \quad \textbf{else}\text{: } \bot \\
& & \textbf{else}\text{: } \bot
\end{array}$$

Though we have not provided details here, Coq's co-inductive types come with some quite stringent restrictions, designed to prevent unsound interactions with proofs. Our definition of program denotation is designed to satisfy those restrictions. The main idea here is that functions defined as *co-fixed points* must be "sufficiently productive"; for our trace example, a co-recursive call may only be made after at least one token has already been added to the stream in the current call. This restriction is the reason for including the seemingly information-free stars in our definition of traces. As we have succeeded in drafting a definition that satisfies this requirement, we are rewarded with a function that is not just an arbitrary relational specification, but rather a *program* that Coq is able to execute, resulting in a Haskell-style lazy list.

2.6 Flat

We are now almost ready to move to assembly language. Our last stop before then is the Flat language, which differs from assembly only in maintaining the abstract view of the heap as a list of tagged records. We do away with variables and move instead to an infinite bank of registers. Function signatures are expressed as maps from natural numbers to types, and Flat programs are responsible for shifting registers around to compensate for the removal of the built-in stack discipline that variable binding provided.

$$\begin{array}{rrcl}
\text{Types} & \tau & ::= & \mathsf{N} \mid \mathsf{ref} \mid \Delta \to \mathbb{N} \\
\text{Typings} & \Delta & = & \mathbb{N} \to \tau \\
\text{Registers} & r & & \\
\text{Operands} & o & ::= & n \mid r \mid \langle n \rangle \mid \mathsf{new}\ \langle \vec{r} \rangle \mid \pi_i r \\
\text{Terms} & e & ::= & r := o; e \mid \mathsf{jump}\ r \\
\text{Programs} & p & ::= & \mathsf{let}\ e\ \mathsf{in}\ p \mid e
\end{array}$$

In the style of Typed Assembly Language, to each static point in a Flat program we associate a typing Δ expressing our expectations of register types whenever that point is reached dynamically. It is crucial that we keep this typing information, since we will use it to create garbage collector root tables in the next and final transformation.

Since there is no need to encode any variable binding for Flat, its denotational semantics is entirely routine. The only exception is that we re-use the trace semantics from Alloc.

2.7 Asm

Our idealized assembly language Asm was already introduced in Section 1.1. We have by now already provided the main ingredients needed to give it a denotational semantics. We re-use the trace-based approach from the last two languages. The difference we must account for is the shift from abstract to concrete heaps, as well as the fact that Asm is parametric in a runtime system.

We should make it clear that we have not verified any runtime system or garbage collector, but only stated conditions that they ought to satisfy and proved that those conditions imply the correctness of our compiler. Recent work on formal certification of garbage collectors [MSLL07] gives us hope that the task we have omitted is not insurmountable.

In more detail, our formalization of Asm starts with the definition of the domain $\mathbb{H}$ of concrete heaps:

$$\mathbb{H} \quad = \quad \mathbb{N} \to \mathbb{N}$$

A runtime system provides new and read operations. The read operation is simpler:

$$\mathsf{read} \quad : \quad \mathbb{H} \times \mathbb{N} \times \mathbb{N} \to \mathbb{N}$$

For a heap, a pointer to a heap-allocated record, and a constant field offset within that record, read should return that field's current value. Runtime systems may make different decisions on concrete layout of records. For instance, there are several ways of including information on which fields are pointers. Note that read is an arbitrary Coq function, not a sequence of assembly instructions, which should let us reason about many different runtime system design decisions within our framework.

The new operation is more complicated, as it is designed to facilitate garbage collector operation.

$$\begin{array}{rcl}
\mathsf{new} & : & \mathbb{H} \times \mathsf{list}\ \mathbb{N} \times \mathsf{list}\ \mathbb{N} \\
& & \to \mathbb{H} \times \mathsf{list}\ \mathbb{N} \times \mathbb{N}
\end{array}$$

Its arguments are the current heap, the values to use to initialize the fields of the record being allocated, and the values of all live registers holding pointer values. The idea is that, in the course of fulfilling the allocation request, the runtime system may rearrange memory however it likes, so long as things afterward look the same as before to any type-safe program, from the perspective of the live registers. A return value of new gives the modified heap, a fresh set of values for the pointer-holding registers (i.e., the garbage collection roots, which may have been moved by a copying collector), and a pointer to the new record in the new heap.

To state the logical conditions imposed on runtime systems, we will need to make a definition based on the abstract heap model of earlier intermediate languages:

DEFINITION 1 (Pointer isomorphism). *We say that pointers $p_1, p_2 \in \mathbb{N}$ are isomorphic with respect to abstract heaps $m_1, m_2 \in \mathbb{M}$ iff:*

1. *The p_1th record of m_1 and p_2th record of m_2 have the same number of fields.*
2. *If the ith field of the p_1th record of m_1 is tagged* Untraced*, then the ith field of the p_2th record of m_2 is also tagged* Untraced*, and the two fields have the same value.*
3. *If the ith field of the p_1th record of m_1 is tagged* Traced*, then the ith field of the p_2th record of m_2 is also tagged* Traced*, and the two fields contain isomorphic pointers.*

The actual definition is slightly more involved but avoids this unqualified self-reference.

To facilitate a parametric translation soundness proof, a candidate runtime system is required to provide a concretization function:

$$\gamma \quad : \quad \mathbb{M} \to \mathbb{H}$$

For an abstract heap m, $\gamma(m)$ is the concrete heap with which the runtime system's representation conventions associate it. We also overload γ to stand for a different function for concretizing pointers. We abuse notation by applying γ to various different kinds of objects, where it's clear how they ought to be converted from abstract to concrete in terms of the two primary γ translations; and, while some of these γ functions really need to take the abstract heap as an additional argument, we omit it for brevity where the proper value is clear from context. The runtime system must come with proofs that its new and read operations satisfy the appropriate commutative diagrams with these concretization functions.

To give a specific example, we show the more complicated condition between the two operations, that for new.

THEOREM 2 (Correctness of a new implementation). *For any abstract heap m, tagged record field values $\vec{v}$ (each in $\mathbb{C}$), and register root set values $\vec{rs}$, there exist new abstract heap m', new root values $\vec{rs}'$, and new record address a such that:*

1. $\mathit{new}(\gamma(m), \gamma(\vec{v}), \gamma(\vec{rs})) = (\gamma(m'), \gamma(\vec{rs}'), \gamma(a))$
2. *For every pair of values p and p' in $\vec{rs}$ and $\vec{rs}'$, respectively, p and p' are isomorphic with respect to $m \oplus [\vec{v}]$ and m'.*
3. *$|m|$ and a are isomorphic with respect to $m \oplus [\vec{v}]$ and m'.*

To understand the details of the last two conditions, recall that, in the abstract memory model, we allocate new records at the end of the heap, which is itself a list of records. The length of the heap before an allocation gives the proper address for the next record to be allocated.

3. Implementation Strategy Overview

Now that we have sketched the basic structure of our compiler, we move to introducing the ideas behind its implementation in the Coq proof assistant. In this section, we single out the first compiler phase and walk through its implementation and proof at a high level, noting the fundamental challenges that we encounter along the way. We summarize our solution to each challenge and then discuss each in more detail in a later section.

Recall that our first phase is analogous to the first phase of a two-phase CPS transform. We want to translate the Source language to the Linear language.

Of course, before we can begin writing the translation, we need to represent our languages! Thus, our first challenge is to choose a representation of each language using Coq types. Our solution here is based on de Bruijn indices and dependently-typed abstract syntax trees. Our use of dependent typing will ensure that representable terms are free of dangling variable references by encoding a term's free variable set in its type. Moreover, we will *combine typing derivations and terms*, essentially representing each term as its typing derivation. In this way, only *well-typed* terms are representable.

Now we can begin writing the translation from Source to Linear. We have some choices about how to represent translations in Coq, but, with our previous language representation choice, it is quite natural to represent translations as dependently-typed Coq functions. Coq's type system will ensure that, when a translation is fed a well-typed input program, it produces a well-typed output program. We can use Coq's *program extraction* facility to produce OCaml versions of our translations by erasing their dependently-typed parts.

This strategy is very appealing, and it is the one we have chosen, but it is not without its inconveniences. Since we represent terms as their typing derivations, standard operations like bringing a new, unused variable into scope are not no-ops like they are with some other binding representations. These variable operations turn out to correspond to standard theorems about typing contexts. For instance, bringing an unused variable into scope corresponds to a *weakening* lemma. These syntactic functions are tedious to write for each new language, especially when dealing with strong dependent types. Moreover, their implementations have little to do with details of particular languages. As a result, we have been able to create a generic programming system that produces them automatically for arbitrary languages satisfying certain criteria. We not only produce the functions automatically, but we also produce proofs that they commute with arbitrary compositional denotation functions in the appropriate, function-specific senses.

Now assume that we have our translation implemented. Its type ensures that it preserves well-typedness, but we also want to prove that it preserves meaning. What proof technique should we use? The technique of logical relations [Plo73] is the standard for this sort of task. We characterize the relationships between program entities and their compilations using relations defined recursively on type structure. Usual logical relations techniques for denotational semantics translate very effectively into our setting, and we are able to take good advantage of the expressiveness of our meta language CIC in enabling succinct definitions.

With these relations in hand, we reach the point where most pencil-and-paper proofs would say that the rest follows by "routine inductions" of appropriate kinds. Unfortunately, since we want to convince the Coq proof checker, we will need to provide considerably more detail. There is no magic bullet for automating proofs of this sophistication, but we did identify some techniques that worked surprisingly well for simplifying the proof burden. In particular, since our compiler preserves type information, we were able to automate significant portions of our proofs using type-based heuristics. The key insight was the possibility of using *greedy quantifier instantiation*, since the dependent types we use are so particular that most logical quantifiers have domains compatible with just a single subterm of a proof sequent.

4. Representing Typed Languages

We begin our example by representing the target language of its transformation. The first step is easy; we give a standard algebraic datatype definition of the type language shared by Source and Linear.

$$
\begin{aligned}
\mathsf{type} &: \mathsf{set} \\
\mathsf{Nat} &: \mathsf{type} \\
\mathsf{Arrow} &: \mathsf{type} \to \mathsf{type} \to \mathsf{type}
\end{aligned}
$$

Things get more interesting when we go to define our term languages. We want to use dependent types to ensure that only terms with object language typing derivations are representable, so we make our classes of operands and terms indexed type families whose indices tells us their object language types. Coq's expressive dependent type system admits simple implementations of this idea. Its *inductive types* are a generalization of the generalized algebraic datatypes [She04] that have become popular recently.

We represent variables with de Bruijn indices. In other words, a variable is represented as a natural number counting how many binders outward in lexical scoping order one must search to find its binder. Since we are using dependent typing, variables are more than just natural numbers. We represent our de Bruijn contexts as lists of types, and variables are in effect constructive proofs that a particular type is found in a particular context. A generic type family var encapsulates this functionality.

We define operands and terms as two mutually-inductive Coq types. Below, the standard notation Π is used for a dependent function type. The notation $\Pi x : \tau_1, \tau_2$ denotes a function from τ_1 to τ_2, where the variable x is *bound* in the range type τ_2, indicating that the function's result type may depend on the *value* of its argument. We elide type annotations from Π types where they are clear from context.

$$
\begin{aligned}
\mathsf{lprimop} &: \mathsf{list\ type} \to \mathsf{type} \to \mathsf{set} \\
\mathsf{LConst} &: \Pi\Gamma, \mathbb{N} \to \mathsf{lprimop}\ \Gamma\ \mathsf{Nat} \\
\mathsf{LVar} &: \Pi\Gamma, \Pi\tau, \mathsf{var}\ \Gamma\ \tau \to \mathsf{lprimop}\ \Gamma\ \tau \\
\mathsf{LLam} &: \Pi\Gamma, \Pi\tau_1, \Pi\tau_2, \mathsf{lterm}\ (\tau_1 :: \Gamma)\ \tau_2 \\
&\quad \to \mathsf{lprimop}\ \Gamma\ (\mathsf{Arrow}\ \tau_1\ \tau_2)
\end{aligned}
$$

```
Inductive sty : Set :=
  | SNat : sty
  | SArrow : sty -> sty -> sty.

Inductive sterm : list sty -> sty -> Set :=
  | SVar : forall G t,
    Var G t
    -> sterm G t
  | SLam : forall G dom ran,
    sterm (dom :: G) ran
    -> sterm G (SArrow dom ran)
  | SApp : forall G dom ran,
    sterm G (SArrow dom ran)
    -> sterm G dom
    -> sterm G ran
  | SConst : forall G, nat -> sterm G SNat.

Fixpoint styDenote (t : sty) : Set :=
  match t with
    | SNat => nat
    | SArrow t1 t2 => styDenote t1 -> styDenote t2
  end.

Fixpoint stermDenote (G : list sty) (t : sty)
    (e : sterm G t) {struct e}
  : Subst styDenote G -> styDenote t :=
  match e in (sterm G t)
      return (Subst styDenote G -> styDenote t) with
    | SVar _ _ v => fun s =>
      VarDenote v s
    | SLam _ _ _ e' => fun s =>
      fun x => stermDenote e' (SCons x s)
    | SApp _ _ _ e1 e2 => fun s =>
      (stermDenote e1 s) (stermDenote e2 s)
    | SConst _ n => fun _ => n
  end.
```

Figure 1. Coq source code of Source syntax and semantics

$$\begin{array}{rcl}
\mathsf{lterm} & : & \mathsf{list\ type} \rightarrow \mathsf{type} \rightarrow \mathsf{set} \\
\mathsf{LLet} & : & \Pi\Gamma, \Pi\tau_1, \Pi\tau_2, \mathsf{lprimop}\ \Gamma\ \tau_1 \\
 & & \rightarrow \mathsf{lterm}\ (\tau_1 :: \Gamma)\ \tau_2 \rightarrow \mathsf{lterm}\ \Gamma\ \tau_2 \\
\mathsf{LThrow} & : & \Pi\Gamma, \Pi\tau, \mathsf{lprimop}\ \Gamma\ \tau \rightarrow \mathsf{lterm}\ \Gamma\ \tau \\
\mathsf{LApp} & : & \Pi\Gamma, \Pi\tau_1, \Pi\tau_2, \Pi\tau_3, \mathsf{var}\ \Gamma\ (\mathsf{Arrow}\ \tau_1\ \tau_2) \\
 & & \rightarrow \mathsf{var}\ \Gamma\ \tau_1 \rightarrow \mathsf{var}\ \Gamma\ (\mathsf{Arrow}\ \tau_2\ \tau_3) \\
 & & \rightarrow \mathsf{lterm}\ \Gamma\ \tau_3
\end{array}$$

In Coq, all constants are defined in the same syntactic class, as opposed to using separate type and term languages. Thus, for instance, lprimop and LConst are defined "at the same level," though the type of the former tells us that it is "a type."

As an example of the encoding, the identity term $\lambda x : \mathsf{N}.\ \mathsf{throw}\ \langle x\rangle$ is represented as:

$$\mathsf{LLam}\ []\ \mathsf{Nat}\ \mathsf{Nat}\ (\mathsf{LThrow}\ [\mathsf{Nat}]\ \mathsf{Nat}\ (\mathsf{LVar}\ [\mathsf{Nat}]\ \mathsf{Nat}\ \mathsf{First}))$$

where First is a constructor of var indicating the lexically innermost variable.

To give an idea of the make-up of our real implementation, we will also provide some concrete Coq code snippets throughout this article. The syntax and static and dynamic semantics of our source language are simple enough that we can show them here in their entirety in Figure 1. Though Coq syntax may seem strange at first to many readers, the structure of these definitions really mirrors their informal counterparts exactly. These snippets are only meant to give a flavor of the project. We describe in Section 10 how to obtain the complete project source code, for those who want a more in-depth treatment.

5. Representing Transformations

We are able to write the linearization transformation quite naturally using our de Bruijn terms. We start this section with a less formal presentation of it.

We first define an auxiliary operation $e_1 \overset{u}{\bullet} e_2$ that, given two linear terms e_1 and e_2 and a variable u free in e_2, returns a new linear term equivalent to running e_1 and throwing its result to e_2 by binding that result to u.

$$\begin{array}{rcl}
(\mathsf{let}\ y = o\ \mathsf{in}\ e_1) \overset{u}{\bullet} e_2 & = & \mathsf{let}\ y = o\ \mathsf{in}\ (e_1 \overset{u}{\bullet} e_2) \\
(\mathsf{throw}\ \langle o\rangle) \overset{u}{\bullet} e & = & \mathsf{let}\ u = o\ \mathsf{in}\ e \\
(x\ y\ z) \overset{u}{\bullet} e & = & \mathsf{let}\ f = (\lambda v.\ \mathsf{let}\ g = (\lambda u.\ e)\ \mathsf{in}\ z\ v\ g) \\
 & & \mathsf{in}\ x\ y\ f
\end{array}$$

Now we can give the linearization translation itself.

$$\begin{array}{rcl}
\lfloor n\rfloor & = & \mathsf{throw}\ \langle n\rangle \\
\lfloor x\rfloor & = & \mathsf{throw}\ \langle x\rangle \\
\lfloor e_1\ e_2\rfloor & = & \lfloor e_1\rfloor \overset{u}{\bullet} (\lfloor e_2\rfloor \overset{v}{\bullet} (\mathsf{let}\ f = (\lambda x.\ \mathsf{throw}\ \langle x\rangle)\ \mathsf{in}\ u\ v\ f))) \\
\lfloor \lambda x : \tau.\ e\rfloor & = & \mathsf{throw}\ \langle \lambda x : \tau.\ \lfloor e\rfloor\rangle
\end{array}$$

This translation can be converted rather directly into Coq recursive function definitions. The catch comes as a result of our strong dependent types for program syntax. The Coq type checker is not able to verify the type-correctness of some of the clauses above.

For a simple example, let us focus on part of the linearization case for applications. There we produce terms of the form $\lfloor e_1\rfloor \overset{u}{\bullet} (\lfloor e_2\rfloor \overset{v}{\bullet} \ldots)$. The term e_2 originally occurred in some context Γ. However, here $\lfloor e_2\rfloor$, the compilation of e_2, is used in a context formed by adding an additional variable u to Γ. We know that this transplantation is harmless and ought to be allowed, but the Coq type checker flags this section as a type error.

We need to perform an explicit coercion that adjusts the de Bruijn indices in $\lfloor e_2\rfloor$. The operation we want corresponds to a weakening lemma for our typing judgment: "If $\Gamma \vdash e : \tau$, then $\Gamma, x : \tau' \vdash e : \tau$ when x is not free in e." Translating this description into our formalization with inductive types from the last section, we want a function:

$$\mathsf{weakenFront} \quad : \quad \Pi\Gamma, \Pi\tau, \mathsf{lterm}\ \Gamma\ \tau \rightarrow \Pi\tau', \mathsf{lterm}\ (\tau' :: \Gamma)\ \tau$$

It is possible to write a custom weakening function for linearized terms, but nothing about the implementation is specific to our language. There is a generic recipe for building weakening functions, based only on an understanding of where variable binders occur. To keep our translations free of such details, we have opted to create a generic programming system that builds these syntactic helper functions for us. It is the subject of the next section.

When we insert these coercions where needed, we have a direct translation of our informal compiler definition into Coq code. Assuming for now that the coercion functions have already been generated, we can give as an example of a real implementation the Coq code for the main CPS translation, in Figure 2. We lack the space to describe the code in detail, but we point out that the `Next` and `First` constructors are used to build de Bruijn variables in unary form. `compose` is the name of the function corresponding to our informal $\overset{u}{\bullet}$ operator. `Lterm` is a module of helper functions gen-

```
Fixpoint cps (G : list sty) (t : sty)
    (e : sterm G t) {struct e}
  : lterm G t :=
  match e in (sterm G t) return (lterm G t) with
    | SVar _ _ v => LThrow (LVar v)
    | SConst _ n => LThrow (LConst _ n)
    | SLam _ _ _ e' => LThrow (LLam (cps e'))
    | SApp _ _ _ e1 e2 =>
      compose (cps e1)
      (compose (Lterm.weakenFront _ (cps e2))
         (LBind
          (LLam (LThrow (LVar First)))
          (LApply
            (Next (Next First))
            (Next First)
            First)))
  end.
```

Figure 2. Coq source code of the main CPS translation

erated automatically, and it includes the coercion weakenFront described earlier. Let us, then, turn to a discussion of the generic programming system that produces it.

6. Generic Syntactic Functions

A variety of these syntactic helper functions come up again and again in formalization of programming languages. We have identified a few primitive functions that seemed to need per-language implementations, along with a number of derived functions that can be implemented parametrically in the primitives. Our generic programming system writes the primitive functions for the user and then automatically instantiates the parametric derived functions. We present here a catalogue of the functions that we found to be important in this present work. All operate over some type family term parameterized by types from an arbitrary language.

The first primitive is a generalization of the weakening function from the last section. We modify its specification to allow insertion into any position of a context, not just the beginning. $\oplus$ denotes list concatenation, and it and single-element concatenation :: are right associative at the same precedence level.

$$\begin{aligned}\mathsf{weaken} \;&:\; \Pi\Gamma_1, \Pi\Gamma_2, \Pi\tau, \mathsf{term}\ (\Gamma_1 \oplus \Gamma_2)\ \tau \\ &\rightarrow \Pi\tau', \mathsf{term}\ (\Gamma_1 \oplus \tau' :: \Gamma_2)\ \tau\end{aligned}$$

Next is elementwise permutation. We swap the order of two adjacent context elements.

$$\begin{aligned}\mathsf{permute} \;&:\; \Pi\Gamma_1, \Pi\Gamma_2, \Pi\tau_1, \Pi\tau_2, \Pi\tau, \mathsf{term}\ (\Gamma_1 \oplus \tau_1 :: \tau_2 :: \Gamma_2)\ \tau \\ &\rightarrow \mathsf{term}\ (\Gamma_1 \oplus \tau_2 :: \tau_1 :: \Gamma_2)\ \tau\end{aligned}$$

We also want to calculate the free variables of a term. Here we mean those that actually appear, not just those that are in scope. This calculation and related support functions are critical to efficient closure conversion for any language. We write $\mathcal{P}(\Gamma)$ to denote the type of subsets of the bindings in context Γ.

$$\mathsf{freeVars} \;:\; \Pi\Gamma, \Pi\tau, \mathsf{term}\ \Gamma\ \tau \rightarrow \mathcal{P}(\Gamma)$$

Using the notion of free variables, we can define *strengthening*, which removes unused variable bindings from a context. This operation is also central to closure conversion. An argument of type $\Gamma_1 \subseteq \Gamma_2$ denotes a constructive proof that every binding in Γ_1 is also in Γ_2.

$$\begin{aligned}\mathsf{strengthen} \;&:\; \Pi\Gamma, \Pi\tau, \Pi e : \mathsf{term}\ \Gamma\ \tau, \Pi\Gamma', \\ &\mathsf{freeVars}\ \Gamma\ \tau\ e \subseteq \Gamma' \rightarrow \mathsf{term}\ \Gamma'\ \tau\end{aligned}$$

We have found these primitive operations to be sufficient for easing the burden of manipulating our higher-level intermediate languages. The derived operations that our system instantiates are: adding multiple bindings to the middle of a context and adding multiple bindings to the end of a context, based on weaken; and moving a binding from the front to the middle or from the middle to the front of a context and swapping two adjacent multi-binding sections of a context, derived from permute.

6.1 Generic Correctness Proofs

Writing these boilerplate syntactic functions is less than half of the challenge when it comes to proving semantics preservation. The semantic properties of these functions are crucial to enabling correctness proofs for the transformations that use them. We also automate the generation of the correctness proofs, based on an observation about the classes of semantic properties we find ourselves needing to verify for them. The key insight is that we only require that each function *commute with any compositional denotation function* in a function-specific way.

An example should illustrate this idea best. Take the case of the weakenFront function for our Source language. Fix an arbitrary denotation function $[\![\cdot]\!]$ for source terms. We claim that the correctness of a well-written compiler will only depend on the following property of weakenFront, written with informal notation: For any Γ, τ, and e with $\Gamma \vdash e : \tau$, we have for any τ', substitution σ for the variables of Γ, and value v of type $[\![\tau']\!]$, that:

$$[\![\mathsf{weakenFront}\ \Gamma\ \tau\ e\ \tau']\!](\sigma, v) = [\![e]\!]\sigma$$

We shouldn't need to know many specifics about $[\![\cdot]\!]$ to deduce this theorem. In fact, it turns out that all we need is the standard property used to judge suitability of denotational semantics, compositionality. In particular, we require that there exist functions f_{const}, f_{var}, f_{app}, and f_{lam} such that:

$$\begin{aligned}[\![n]\!]\sigma &= f_{const}(n) \\ [\![x]\!]\sigma &= f_{var}(\sigma(x)) \\ [\![e_1\ e_2]\!]\sigma &= f_{app}([\![e_1]\!]\sigma, [\![e_2]\!]\sigma) \\ [\![\lambda x : \tau.\ e]\!]\sigma &= f_{lam}(\lambda x : [\![\tau]\!].\ [\![e]\!](\sigma, x))\end{aligned}$$

Our generic programming system introspects into user-supplied denotation function definitions and extracts the functions that witness their compositionality. Using these functions, it performs automatic proofs of generic theorems like the one above about weakenFront. Every other generated syntactic function has a similar customized theorem statement and automatic strategy for proving it for compositional denotations.

Though space limits prevent us from providing more detail here, we mention that our implementation is interesting in that the code and proof generation themselves are implemented almost entirely inside of Coq's programming language, using dependent types to ensure their soundness. We rely on the technique of reflective proofs [Bou97] to enable Coq programs to manipulate other Coq programs in a type-safe manner, modulo some small "proof hints" provided from outside the logic.

7. Representing Logical Relations

We now turn our attention to formulating the correctness proofs of compiler phases, again using the linearization phase as our example. We are able to give a very simple logical relations argument for this phase, since our meta language CIC is sufficiently expressive to encode naturally the features of both source and target languages. The correctness theorem that we want looks something like the following, for some appropriate type-indexed relation $\simeq$. For disambiguation purposes, we write $[\![\cdot]\!]^S$ for Source language denotations and $[\![\cdot]\!]^L$ for Linear denotations.

```
Fixpoint val_lr (t : sty) : styDenote t
  -> ltyDenote t -> Prop :=
  match t
    return (styDenote t -> ltyDenote t -> Prop) with
    | SNat => fun n1 n2 =>
      n1 = n2
    | SArrow t1 t2 => fun f1 f2 =>
      forall x1 x2, val_lr t1 x1 x2
        -> forall (k : styDenote t2 -> nat),
          exists thrown, f2 x2 k = k thrown
            /\ val_lr t2 (f1 x1) thrown
  end.
```

Figure 3. Coq source code for the first-stage CPS transform's logical relation

THEOREM 3 (Correctness of linearization). ***For*** *Source term* e *such that* $\Gamma \vdash e : \tau$, ***if*** *we have valuations* σ^S *and* σ^L *for* $[\![\Gamma]\!]^S$ *and* $[\![\Gamma]\!]^L$*, respectively,* ***such that*** *for every* $x : \tau' \in \Gamma$*, we have* $\sigma^S(x) \simeq_{\tau'} \sigma^L(x)$*,* ***then*** $[\![e]\!]^S \sigma^S \simeq_\tau [\![\lfloor e \rfloor]\!]^L \sigma^L$.

This relation for values turns out to satisfy our requirements:

$$\begin{array}{rcl} n_1 \simeq_{\mathsf{N}} n_2 & = & n_1 = n_2 \\ f_1 \simeq_{\tau_1 \to \tau_2} f_2 & = & \forall x_1 : [\![\tau_1]\!]^S, \forall x_2 : [\![\tau_1]\!]^L, x_1 \simeq_{\tau_1} x_2 \\ & & \to \exists v : [\![\tau_2]\!]^L, \forall k : [\![\tau_2]\!]^L \to \mathbb{N}, \\ & & \quad f_2\ x_2\ k = k\ v \land f_1\ x_1 \simeq_{\tau_2} v \end{array}$$

We have a standard logical relation defined by recursion on the structure of types. $e_1 \simeq_\tau e_2$ means that values e_1 of type $[\![\tau]\!]^S$ and e_2 of type $[\![\tau]\!]^L$ are equivalent in a suitable sense. Numbers are equivalent if and only if they are equal. Source function f_1 and linearized function f_2 are equivalent if and only if for every pair of arguments x_1 and x_2 related at the domain type, there exists some value v such that f_2 called with x_2 and a continuation k always throws v to k, and the result of applying f_1 to x_1 is equivalent to v at the range type.

The suitability of particular logical relations to use in compiler phase correctness specifications is hard to judge individually. We know we have made proper choices when we are able to compose all of our correctness results to form the overall compiler correctness theorem. It is probably also worth pointing out here that our denotational semantics are not *fully abstract*, in the sense that our target domains "allow more behavior" than our object languages ought to. For instance, the functions k quantified over by the function case of the $\simeq$ definition are drawn from the full Coq function space, which includes all manner of complicated functions relying on inductive and co-inductive types. The acceptability of this choice for our application is borne out by our success in using this logical relation to prove a final theorem whose statement does not depend on such quantifications.

Once we are reconciled with that variety of caveat, we find that Coq provides quite a congenial platform for defining logical relations for denotational semantics. The $\simeq$ definition can be transcribed quite literally, as witnessed by Figure 3. The set of all logical propositions in Coq is just another type `Prop`, and so we may write recursive functions that return values in it. Contrast our options here with those associated with proof assistants like Twelf [PS99], for which formalization of logical relations has historically been challenging.

8. Proof Automation

Now that we have the statement of our theorem, we need to produce a formal proof of it. In general, our proofs will require significant manual effort. As anyone who has worked on computerized proofs can tell you, time-saving automation machinery is extremely welcome. In proving the correctness theorems for our compiler, we were surprised to find that a very simple automation technique is very effective on large classes of proof goals that appear. The effectiveness of this technique has everything to do with the combination of typed intermediate languages and our use of dependent types to represent their programs.

As an example, consider this proof obligation that occurs in the correctness proof for linearization.

- **Know:** $\forall \sigma^S, \forall \sigma^L, \sigma^S \simeq_\Gamma \sigma^L \to \exists v : [\![\tau_1 \to \tau_2]\!]^L, \forall k : [\![\tau_1 \to \tau_2]\!]^L \to \mathbb{N}, [\![\lfloor e_1 \rfloor]\!]^L \sigma^L\ k = k\ v \land [\![e_1]\!]^S \sigma^S \simeq_{\tau_1 \to \tau_2} v$
- **Know:** $\forall \sigma^S, \forall \sigma^L, \sigma^S \simeq_\Gamma \sigma^L \to \exists v : [\![\tau_1]\!]^L, \forall k : [\![\tau_1]\!]^L \to \mathbb{N}, [\![\lfloor e_2 \rfloor]\!]^L s^L\ k = k\ v \land [\![e_2]\!]^S \sigma^S \simeq_{\tau_1} v$
- **Must prove:** $\forall \sigma^S, \forall \sigma^L, \sigma^S \simeq_\Gamma \sigma^L \to \exists v : [\![\tau_2]\!]^L, \forall k : [\![\tau_2]\!]^L \to \mathbb{N}, [\![\lfloor e_1\ e_2 \rfloor]\!]^L \sigma^L\ k = k\ v \land [\![e_1\ e_2]\!]^S \sigma^S \simeq_{\tau_2} v$

It is safe to simplify the goal by moving all of the universal quantifiers and implications that begin it into our proof context as new bound variables and assumptions; this rearrangement cannot alter the provability of the goal. Beyond that, Coq's standard automation support is stuck. However, it turns out that we can do much better for goals like this one, based on *greedy quantifier instantiation*. Traditional automated theorem provers spend most of their intelligence in determining how to *use* universally-quantified facts and *prove* existentially-quantified facts. When quantifiers range over infinite domains, many such theorem-proving problems are both undecidable and difficult in practice.

However, examining our goal above, we notice that it has a very interesting property: *every* quantifier has a rich type depending on some object language type. Moreover, for any of these types, *exactly one subterm of proof state* (bound variables, assumptions, and goal) that has that type ever appears at any point during the proving process! This observation makes instantiation of quantifiers extremely easy: instantiate any universal assumption or existential goal with *the first properly-typed proof state subterm that appears*.

For instance, in the example above, we had just moved the variables σ^S and σ^L and the assumption $\sigma^S \simeq_\Gamma \sigma^L$ into our proof context. That means that we should instantiate the initial σ quantifiers of the two assumptions with these variables and use modus ponens to access the conclusions of their implications, by way of our new assumption. This operation leaves both assumptions at existential quantifiers, so we eliminate these quantifiers by adding fresh variables and new assumptions about those variables. The types of the k quantifiers that we reach now don't match any subterms in scope, so we stop here.

Now it's time for a round of rewriting, using rules added by the human user to a hint database. We use all of the boilerplate syntactic function soundness theorems that we generated automatically as left-to-right rewrite rules, applying them in the goal until no further changes are possible. Using also a rewrite theorem expressing the soundness of the $\overset{u}{\bullet}$ composition operation, this process simplifies the goal to a form with a subterm $[\![\lfloor e_1 \rfloor]\!]^L\ s^L(\lambda x : [\![\tau_1 \to \tau_2]\!]^L. \ldots)$. This lambda term has the right type to use as an instantiation for the universally-quantified k in the first assumption, so, by our greedy heuristic, we make that instantiation. We perform "safe" propositional simplifications on the newly-exposed part of that assumption and continue.

Iterating this heuristic, we discharge the proof obligation completely, with no human intervention. Our very naive algorithm has succeeded in "guessing" all of the complicated continuations needed for quantifier instantiations, simply by searching for

```
(* State the lemma characterizing the effect of
 * the CPS'd term composition operator. *)
Lemma compose_sound : forall (G : list sty)
  (t : sty) (e : lterm G t)
  (t' : sty) (e' : lterm (t :: G) t') s
  (k : _ -> result),
  ltermDenote (compose e e') s k
  = ltermDenote lin s
    (fun x => ltermDenote lin' (SCons x s) k).
  induction e;    (* We prove it by induction on
                   * the structure of
                   * the term e. *)
    equation_tac. (* A generic rewriting
                   * procedure handles
                   * the resulting cases. *)
Qed.

(* ...omitted code to add compose_sound to our
 * rewriting hint base... *)

(* State the theorem characterizing soundness of
 * the CPS translation. *)
Theorem cps_sound : forall G t (e : sterm G t),
  exp_lr e (cps e).
  unfold exp_lr; (* Expand the definition of the
                  * logical relation on
                  * expressions. *)
    induction e; (* Proceed by induction on the
                  * structure of the term e. *)
    lr_tac.      (* Use the generic greedy
                  * instantiation procedure to
                  * discharge the subgoals. *)
Qed.
```

Figure 4. Snippets of the Coq proof script for the CPS correctness theorem

properly-typed subterms of the proof state. In fact, this same heuristic discharges all of the cases of the linearization soundness proof, once we've stocked the rewrite database with the appropriate syntactic simplifications beforehand. To prove this theorem, all the user needs to do is specify which induction principle to use and run the heuristic on the resulting subgoals.

We have found this approach to be very effective on high-level typed languages in general. The human prover's job is to determine the useful syntactic properties with which to augment those proved generically, prove these new properties, and add them to the rewrite hint database. With very little additional effort, the main theorems can then be discharged automatically. Unfortunately, our lower-level intermediate languages keep less precise type information, so greedy instantiation would make incorrect choices too often to be practical. Our experience here provides a new justification in support of type-preserving compilation.

Figure 4 shows example Coq code for proving the CPS correctness theorem, with a few small simplifications made for space reasons.

9. Further Discussion

Some other aspects of our formalization outside of the running example are worth mentioning. We summarize them in this section.

9.1 Logical Relations for Closure Conversion

Formalizations of closure conversion and its correctness, especially those using operational semantics, often involve existential types and other relatively complicated notions. In our formalization, we are able to use a surprisingly simple logical relation to characterize closure conversion. It relates denotations from the CPS and CC languages, indicated with superscripts P and C, respectively. We lift various definitions to vectors in the usual way.

$$
\begin{aligned}
n_1 \simeq_{\mathbb{N}} n_2 &= n_1 = n_2 \\
f_1 \simeq_{\vec{\tau} \to \mathbb{N}} f_2 &= \forall \vec{x}_1 : [\![\vec{\tau}]\!]^P, \forall x_2 : [\![\vec{\tau}]\!]^C, x_1 \simeq_{\vec{\tau}} x_2 \\
&\quad \to f_1\ x_1 \simeq_{\tau_2} f_2\ x_2
\end{aligned}
$$

This relation is almost identical to the most basic logical relation for simply-typed lambda calculus! The secret is that our meta language CIC "has native support for closures." That is, Coq's function spaces already incorporate the appropriate reduction rules to capture free variables, so we don't need to mention this process explicitly in our relation.

In more detail, the relevant denotations of CC types are:

$$
\begin{aligned}
[\![\vec{\tau} \to \mathbb{N}]\!] &= [\![\tau_1]\!] \times \ldots \times [\![\tau_n]\!] \to \mathbb{N} \\
[\![\vec{\tau}^1 \times \vec{\tau}^2 \to \mathbb{N}]\!] &= ([\![\tau_1^1]\!] \times \ldots \times [\![\tau_n^1]\!]) \to ([\![\tau_1^2]\!] \times \ldots \times [\![\tau_m^2]\!]) \to \mathbb{N}
\end{aligned}
$$

Recall that the second variety of type shown is the type of code pointers, with the additional first list of parameters denoting the expected environment type. A CPS language lambda expression is compiled into a packaging of a closure using a pointer to a fresh code block. That code block will have some type $\vec{\tau}_1 \times \vec{\tau}_2 \to \mathbb{N}$. The denotation of this type is some meta language type $T_1 \to T_2 \to \mathbb{N}$. We perform an immediate partial application to an environment formed from the relevant free variables. This environment's type will have denotation T_1. Thus, the partial application's type has denotation $T_2 \to \mathbb{N}$, making it compatible with our logical relation. The effect of the closure packaging operation has been "hidden" using one of the meta language's "closures" formed by the partial application.

9.2 Explicating Higher-Order Control Flow

The translation from CC to Alloc moves from a higher-order, terminating language to a first-order language that admits non-termination. As a consequence, the translation correctness proof must correspond to some explanation of why CC's particular brand of higher-order control flow leads to termination, and why the resulting first-order program returns an identical answer to the higher-order original.

The basic proof technique has two main pieces. First, the logical relation requires that any value with a code type terminates with the expected value when started in a heap and variable environment where the same condition holds for values that should have code type. Second, we take advantage of the fact that function definitions occur in dependency order in CC programs. Our variable binding restrictions enforce a lack of cycles via dependent types. We can prove by induction on the position of a code body in a program that any execution of it in a suitable starting state terminates with the correct value. Any code pointer involved in the execution either comes earlier in the program, in which case we have its correctness by the inductive hypothesis; or the code pointer has been retrieved from a variable or heap slot, in which case its safety follows by an induction starting from our initial hypothesis and tracking the variable bindings and heap writes made by the program.

9.3 Garbage Collection Safety

The soundness of the translation from Flat to Asm depends on a delicate safety property of well-typed Flat programs. It is phrased in terms of the register typings we have available at every program point, and it depends on the definition of pointer isomorphism we gave in Section 2.7.

THEOREM 4 (Heap rearrangement safety). *For any register typing Δ, abstract heaps m_1 and m_2, and register files R_1 and R_2, if:*

1. *For every register r with $\Delta(r) = \mathsf{ref}$, $R_1(r)$ is isomorphic to $R_2(r)$ with respect to m_1 and m_2.*
2. *For every register r with $\Delta(r) \neq \mathsf{ref}$, $R_1(r) = R_2(r)$.*

and p is a Flat program such that $\Delta \vdash p$, we have $[\![p]\!]R_1 m_1 = [\![p]\!]R_2 m_2$.

This theorem says that it is safe to rearrange a heap if the relevant roots pointing into it are also rearranged equivalently. Well-typed Flat programs can't distinguish between the old and new situations. The denotations that we conclude are equal in the theorem are traces, so we have that valid Flat programs return identical results (if any) and make the same numbers of function calls in any isomorphic initial states.

The requirements on the runtime system's new operation allow it to modify the heap arbitrarily on each call, so long as the new heap and registers are isomorphic to the old heap and registers. The root set provided as an operand to new is used to determine the appropriate notion of isomorphism, so the heap rearrangement safety theorem is critical to making the correctness proof go through.

9.4 Putting It All Together

With all of the compiler phases proved, we can compose the proofs to form a correctness proof for the overall compiler. We give the formal version of the theorem described informally in the Introduction. We use the notation $T \downarrow n$ to denote that trace T terminates with result n.

THEOREM 5 (Compiler correctness). *Let $m \in \mathbb{H}$ be a heap initialized with a closure for the top-level continuation, let p be a pointer to that closure, and let R be a register file mapping the first register to p. Given a Source term e such that $\cdot \vdash e : \mathsf{N}$, $[\![\lfloor e \rfloor]\!]Rm \downarrow [\![e]\!]()$.*

Even considering only the early phases of the compiler, it is difficult to give a correctness theorem in terms of equality for all source types. For instance, we can't compose parametrically the results for CPS transformation and closure conversion to yield a correctness theorem expressed with an equality between denotations. The problem lies in the lack of full abstraction for our semantics. Instead, we must use an alternate notion of equality that only requires functions to agree at arguments that are denotations of terms. This general notion also appears in the idea of pre-logical relations [HS99], a compositional alternative to logical relations.

10. Implementation

The compiler implementation and documentation are available online at:

`http://ltamer.sourceforge.net/`

We present lines-of-code counts for the different implementation files, including proofs, in Figure 5.

File	LoC
Source	31
...to...	116
Linear	56
...to...	115
CPS	87
...to...	646
CC	185
...to...	1321
Alloc	217
...to...	658
Flat	141
...to...	868
Asm	111

File	LoC
Applicative dictionaries	119
Traces	96
GC safety	741
Overall compiler	119

Figure 5. Project lines-of-code counts

These figures do not include 3520 lines of Coq code from a programming language formalization support library that we have been developing in tandem with the compiler. Additionally, we have 2716 lines of OCaml code supporting generic programming of syntactic support functions and their correctness proofs. Developing these re-usable pieces was the most time-consuming part of our overall effort. We believe we improved our productivity an order of magnitude by determining the right language representation scheme and its library support code, and again by developing the generic programming system. With those pieces in place, implementing the compiler only took about one person-month of work, which we feel validates the general efficacy of our techniques.

It is worth characterizing how much of our implementation must be trusted to trust its outputs. Of course, the Coq system and the toolchain used to compile its programs (including operating system and hardware) must be trusted. Focusing on code specific to this project, we see that, if we want only to certify the behavior of particular output assembly programs, we must trust about 200 lines of code. This figure comes from taking a backwards slice from the statement of any individual program correctness theorem. If we want to believe the correctness of the compiler itself, we must add additionally about 100 lines to include the formalization of the Source language as well.

11. Related Work

Moore produced a verified implementation of a compiler for the Piton language [Moo89] using the Boyer-Moore theorem prover. Piton did not have the higher-order features that make our present work interesting, and proofs in the Boyer-Moore tradition have fundamentally different trustworthiness characteristics than Coq proofs, being dependent on a large reasoning engine instead of a small proof-checking kernel. However, the Piton work dealt with larger and more realistic source and target languages.

The VLISP project [GRW95] produced a Scheme system with a rigorous but non-mechanized proof of correctness. They also made heavy use of denotational semantics, but they dealt with a dynamically-typed source language and so did not encounter many of the interesting issues reported here related to type-preserving compilation.

Semantics preservation proofs have been published before for individual phases of type-preserving compilers, including closure conversion [MMH95]. All of these proofs that we are aware of use operational semantics, forfeiting the advantages of denotational semantics for mechanization that we have shown here.

Recent work has implemented tagless interpreters [PTS02] using generalized algebraic datatypes and other features related to dependent typing. Tagless interpreters have much in common with our approach to denotational semantics in Coq, but we are not aware of any proofs beyond type safety carried out in such settings.

The CompCert project [Ler06] has used Coq to produce a certified compiler for a subset of C. Because of this source language choice, CompCert has not required reasoning about nested variable scopes, first-class functions based on closures, or dynamic allocation. On the other hand, they deal with larger and more realistic source and target languages. CompCert uses non-dependently-typed abstract syntax and operational semantics, in contrast to our use of dependent types and denotational semantics; and we focus more on proof automation.

Many additional pointers to work on compiler verification can be found in the bibliography by Dave [Dav03].

12. Conclusion

We have outlined a non-trivial case study in certification of compilers for higher-order programming languages. Our results lend credence to the suitability of our implementation strategy: the encoding of language syntax and static semantics using dependent types, along with the use of denotational semantics targeting a rich but formalized meta language. We have described how generic programming and proving can be used to ease the development of type-preserving compilers and their proofs, and we have demonstrated how the certification of type-preserving compilers is congenial to automated proof.

We hope to expand these techniques to larger and more realistic source and target languages. Our denotational approach naturally extends to features that can be encoded directly in CIC, including (impredicative) universal types, (impredicative) existential types, lists, and trees. Handling of "effectful" features like nontermination and mutability without first compiling to first-order form (as we've done here in the later stages of the compiler) is an interesting open problem. We also plan to investigate further means of automating compiler correctness proofs and factorizing useful aspects of them into re-usable libraries.

There remains plenty of room to improve the developer experience in using our approach. Programming with dependent types has long been known to be tricky. We implemented our prototype by slogging through the messy details for the small number of features that we've included. In scaling up to realistic languages, the costs of doing so may prove prohibitive. Some recently-proposed techniques for simplifying dependently-typed Coq programming [Soz06] may turn out to be useful.

Coercions between different dependent types appear frequently in the approach we follow. It turns out that effective reasoning about coercions in type theory requires something more than the computational equivalences that are used in systems like CIC. In Coq, this reasoning is often facilitated by adding an axiom describing the computational behavior of coercions. Ad-hoc axioms are a convenient way of extending a proof assistant's logic, but their loose integration has drawbacks. Coq's built-in type equivalence judgment, which is applied automatically during many stages of proof checking, will not take the axioms into account. Instead, they must be applied in explicit proofs. New languages like Epigram [MM04] design their type equivalence judgments to facilitate reasoning about coercions. Future certified compiler projects might benefit from being developed in such environments, modulo the current immaturity of their development tools compared to what Coq offers. Alternatively, it is probably worth experimenting with the transplantation of some of these new ideas into Coq.

Acknowledgments

Thanks to Manu Sridharan and the anonymous referees for helpful comments on drafts of this paper.

References

[ABF+05] Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. Mechanized metatheory for the masses: The POPLMARK challenge. In *Proc. TPHOLs*, pages 50–65, 2005.

[BC04] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer Verlag, 2004.

[Bou97] Samuel Boutin. Using reflection to build efficient and certified decision procedures. In *Proc. STACS*, pages 515–529, 1997.

[Dav03] Maulik A. Dave. Compiler verification: a bibliography. *SIGSOFT Softw. Eng. Notes*, 28(6):2–2, 2003.

[dB72] Nicolas G. de Bruijn. Lambda-calculus notation with nameless dummies: a tool for automatic formal manipulation with application to the Church-Rosser theorem. *Indag. Math.*, 34(5):381–392, 1972.

[Gim95] Eduardo Giménez. Codifying guarded definitions with recursive schemes. In *Proc. TYPES*, pages 39–59. Springer-Verlag, 1995.

[GRW95] Joshua D. Guttman, John D. Ramsdell, and Mitchell Wand. VLISP: A verified implementation of Scheme. *Lisp and Symbolic Computation*, 8(1/2):5–32, 1995.

[HS99] Furio Honsell and Donald Sannella. Pre-logical relations. In *Proc. CSL*, pages 546–561, 1999.

[Ler06] Xavier Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *Proc. POPL*, pages 42–54, 2006.

[MM04] Conor McBride and James McKinna. The view from the left. *J. Functional Programming*, 14(1):69–111, 2004.

[MMH95] Yasuhiko Minamide, Greg Morrisett, and Robert Harper. Typed closure conversion. Technical Report CMU-CS-FOX-95-05, Carnegie Mellon University, 1995.

[Moo89] J. Strother Moore. A mechanically verified language implementation. *J. Automated Reasoning*, 5(4):461–492, 1989.

[MSLL07] Andrew McCreight, Zhong Shao, Chunxiao Lin, and Long Li. A general framework for certifying garbage collectors and their mutators. In *Proc. PLDI*, 2007.

[MWCG99] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. *ACM Trans. Program. Lang. Syst.*, 21(3):527–568, 1999.

[PE88] F. Pfenning and C. Elliot. Higher-order abstract syntax. In *Proc. PLDI*, pages 199–208, 1988.

[Plo73] G. D. Plotkin. Lambda-definability and logical relations. Memorandum SAI-RM-4, University of Edinburgh, 1973.

[Plo75] Gordon D. Plotkin. Call-by-name, call-by-value, and the lambda calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[PS99] Frank Pfenning and Carsten Schürmann. System description: Twelf - a meta-logical framework for deductive systems. In *Proc. CADE*, pages 202–206, 1999.

[PTS02] Emir Pasalic, Walid Taha, and Tim Sheard. Tagless staged interpreters for typed languages. In *Proc. ICFP*, pages 218–229, 2002.

[She04] Tim Sheard. Languages of the future. In *Proc. OOPSLA*, pages 116–119, 2004.

[Soz06] Matthieu Sozeau. Subset coercions in Coq. In *Proc. TYPES*, 2006.

[TMC+96] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: a type-directed optimizing compiler for ML. In *Proc. PLDI*, pages 181–192, 1996.