Formal Compiler Implementation in a Logical Framework*

Jason Hickey, Aleksey Nogin, Adam Granicz, and Brian Aydemir †
California Institute of Technology
1200 E. California Blvd.
Pasadena, CA 91125, USA
{jyh,nogin,granicz,emre}@cs.caltech.edu

ABSTRACT

The task of designing and implementing a compiler can be a difficult and error-prone process. In this paper, we present a new approach based on the use of higher-order abstract syntax and term rewriting in a logical framework. All program transformations, from parsing to code generation, are cleanly isolated and specified as term rewrites. This has several advantages. The correctness of the compiler depends solely on a small set of rewrite rules that are written in the language of formal mathematics. In addition, the logical framework guarantees the preservation of scoping, and it automates many frequently-occurring tasks including substitution and rewriting strategies. As we show, compiler development in a logical framework can be *easier* than in a general-purpose language like ML, in part because of automation, and also because the framework provides extensive support for examination, validation, and debugging of the compiler transformations. The paper is organized around a case study, using the MetaPRL logical framework to compile an ML-like language to Intel x86 assembly. We also present a scoped formalization of x86 assembly in which all registers are immutable.

1. INTRODUCTION

The task of designing and implementing a compiler can be difficult even for a small language. There are many phases in the translation from source to machine code, and an error in any one of these phases can alter the semantics of the generated program. The use of programming languages that provide type safety, pattern matching, and automatic storage management can reduce the compiler's code size and eliminate some common kinds of errors. However, many programming languages that appear well-suited for compiler implementation, like ML [19], still do not address other issues, such as substitution and preservation of scoping in the compiled program.

In this paper, we present an alternative approach, based on the use of higher-order abstract syntax [15, 16] and term rewriting in a general-purpose logical framework. All program transformations, from parsing to code generation, are cleanly isolated and specified as term rewrites. In our system, term rewrites specify an equivalence between two code fragments that is valid in any context. Rewrites are bidirectional and neither imply nor presuppose any particular order of application. Rewrite application is guided by programs in the meta-language of the logical framework.

There are many advantages to using formal rewrites. Program scoping and substitution are managed implicitly by the logical framework; it is not possible to specify a program transformation that modifies the program scope. Perhaps most importantly, the correctness of the compiler is dependent only on the rewriting rules. Programs that guide the application of rewrites do not have to be trusted because they are required to use rewrites for all program transformations. If the rules can be validated against a program semantics, and if the compiler produces a program, that program will be correct relative to those semantics. The role of the guidance programs is to ensure that rewrites are applied in the appropriate order so that the output of the compiler contains only assembly.

The collection of rewrites needed to implement a compiler is small (hundreds of lines of formal mathematics) compared to the entire code base of a typical compiler (often more than tens of thousands of lines of code in a general-purpose programming language). Validation of the former set is clearly easier. Even if the rewrite rules are not validated, it becomes easier to assign accountability to individual rules.

The use of a logical framework has another major advantage that we explore in this paper: in many cases it is *easier* to implement the compiler, for several reasons. The terminology of rewrites corresponds closely to mathematical descriptions frequently used in the literature, decreasing time from concept to implementation. The logical framework provides a great deal of automation, including efficient substitution and automatic α-renaming of variables to avoid capture, as well as a large selection of rewrite strategies to guide the application of program transformations. The com-

†This work was supported in part by the DoD Multidisciplinary University Research Initiative (MURI) program administered by the Office of Naval Research (ONR) under Grant N00014-01-1-0765, the Defense Advanced Research Projects Agency (DARPA), the United States Air Force, the Lee Center, and by NSF Grant CCR 0204193.

*A technical report version of this paper will be available shortly as Caltech technical report caltechCSTR:2003.002 at `http://caltechcstr.library.caltech.edu`

Submitted to ICFP 2003, Uppsala, Sweden.

pilation task is phrased as a theorem-proving problem, and the logical framework provides a means to examine and debug the effects of the compilation process interactively. The facilities for automation and examination establish an environment where it is easy to experiment with new program transformations and extensions to the compiler.

In fairness, formal compilation also has potential disadvantages. The use of higher-order abstract syntax, in which variables in the programming language are represented as variables in the logical language, means that variables cannot be manipulated directly in the formal system; operations that modify the program scope, such as capturing substitution, are difficult if not impossible to express formally. In addition, global program transformations, in which several parts of a program are modified simultaneously, can sometimes be difficult to express with term rewriting.

The most significant impact of using a formal system is that program representations must permit a substitution semantics. Put another way, the logical framework requires the development of *functional* intermediate representations, where heap locations may be mutable, but variables are not. This potentially has a major effect on the formalization of imperative languages, including assembly language, where registers are no longer mutable. This seeming contradiction can be resolved, as we show in the second half of this paper, but it does require a departure from the majority of the literature on compilation methods.

In this paper, we explore these problems and show that formal compiler development is feasible, perhaps easy. We do not specifically address the problem of compiler verification in this paper; our main objective is to develop the models and methods needed during the compilation process. The format of this paper is organized around a case study, where we develop a compiler that generates Intel x86 machine code for an ML-like language using the MetaPRL logical framework [4, 6, 8]. The compiler is fully implemented and online as part of the Mojave research project [7]. This document is generated from the program sources (MetaPRL provides a form of literate programming), and the complete source code is available online at `http://metaprl.org/` as well as in the technical report.

1.1 Organization

The translation from source code to assembly is usually done in three major stages. The parsing phase translates a source file (a sequence of characters) into an abstract syntax tree; the abstract syntax is translated to an intermediate representation; and the intermediate representation is translated to machine code. The reason for the intermediate representation is that many of the transformations in the compiler can be stated abstractly, independent of the source and machine representations.

The language that we are using as an example (see Section 2) is a small language similar to ML [19]. To keep the presentation simple, the language is untyped. However, it includes higher-order and nested functions, and one necessary step in the compilation process is closure conversion, in which the program is modified so that all functions are closed. The high-level outline of the paper is as follows.

- Section 2 parsing
- Section 3 intermediate representation (IR)
- Section 4 Intel x86 assembly code generation
- Section 6 Related work

Before describing each of these stages, we first introduce the terminology and syntax of the formal system in which we define the program rewrites.

1.2 Terminology

All logical syntax is expressed in the language of *terms*. The general syntax of all terms has three parts. Each term has 1) an operator-name (like "sum"), which is a unique name identifying the kind of term; 2) a list of parameters representing constant values; and 3) a set of subterms with possible variable bindings. We use the following syntax to describe terms:

$$\underbrace{opname}_{operator\ name}\ \underbrace{[p_1; \cdots ; p_n]}_{parameters}\underbrace{\{\vec{v}_1.t_1; \cdots ; \vec{v}_m.t_m\}}_{subterms}$$

Displayed form	Term
1	`number[1]{}`
$\lambda x.b$	`lambda[]{ x. b }`
$f(a)$	`apply[] { f; a }`
$x + y$	`sum[]{ x; y }`

A few examples are shown in the table. Numbers have an integer parameter. The `lambda` term contains a binding occurrence: the variable x is bound in the subterm b.

Term rewrites are specified in MetaPRL using second-order variables, which explicitly define scoping and substitution [15]. A second-order variable pattern has the form $v[v_1; \cdots ; v_n]$, which represents an arbitrary term that may have free variables $v_1, \ldots, v_n$. The corresponding substitution has the form $v[t_1; \cdots ; t_n]$, which specifies the simultaneous, capture-avoiding substitution of terms $t_1, \ldots, t_n$ for $v_1, \ldots, v_n$ in the term matched by v. For example, the rule for β-reduction is specified with the following rewrite.

$$\text{[beta]} \quad (\lambda x.v_1[x])\ v_2 \longleftrightarrow v_1[v_2]$$

The left-hand-side of the rewrite is a pattern called the *redex*. The $v_1[x]$ stands for an arbitrary term with free variable x, and v_2 is another arbitrary term. The right-hand-side of the rewrite is called the *contractum*. The second-order variable $v_1[v_2]$ substitutes the term matched by v_2 for x in v_1. A term rewrite specifies that any term that matches the redex can be replaced with the contractum, and vice-versa.

Rewrites that are expressed with second-order notation are strictly more expressive than those that use the traditional substitution notation. The following rewrite is valid in second-order notation.

$$\text{[const]} \quad (\lambda x.v[])\ 1 \longleftrightarrow (\lambda x.v[])\ 2$$

In the context λx, the second-order variable $v[]$ matches only those terms that do not have x as a free variable. No substitution is performed; the β-reduction of both sides of the rewrite yields $v[] \longleftrightarrow v[]$, which is valid reflexively. Normally, when a second-order variable $v[]$ has an empty free-variable set $[]$, we omit the brackets and use the simpler notation v.

MetaPRL is a tactic-based prover that uses OCaml [20] as its meta-language. When a rewrite is defined in MetaPRL, the framework creates an OCaml expression that can be used to apply the rewrite. Code to guide the application of

$$
\begin{array}{rcll}
op & ::= & + \mid - \mid * \mid / & \text{Binary operators} \\
& \mid & = \mid <> \mid < \mid \leq \mid > \mid \geq & \\
& & & \\
e & ::= & \top \mid \bot & \textit{Booleans} \\
& \mid & i & \textit{Integers} \\
& \mid & v & \textit{Variables} \\
& \mid & e \; op \; e & \textit{Binary expressions} \\
& \mid & \lambda v.e & \textit{Anonymous functions} \\
& \mid & \mathbf{if}\; e \;\mathbf{then}\; e \;\mathbf{else}\; e & \textit{Conditionals} \\
& \mid & e[e] & \textit{Subscripting} \\
& \mid & e[e] \leftarrow e & \textit{Assignment} \\
& \mid & e; e & \textit{Sequencing} \\
& \mid & e(e_1, \ldots, e_n) & \textit{Application} \\
& \mid & \mathbf{let}\; v = e \;\mathbf{in}\; e & \textit{Let definitions} \\
& \mid & \mathbf{let\,rec}\; f_1(v_1, \ldots, v_n) = e & \textit{Recursive functions} \\
& & \ldots & \\
& & \mathbf{and}\; f_n(v_1, \ldots, v_n) = e &
\end{array}
$$

Figure 1: Program syntax

rewrites is written in OCaml, using a rich set of primitives provided by MetaPRL. MetaPRL automates the construction of most guidance code; we describe rewrite strategies only when necessary. For clarity, we will describe syntax and rewrites using the displayed forms of terms.

The compilation process is expressed in MetaPRL as a judgment of the form $\Gamma \vdash \mathbf{compilable}(e)$, which states the the program e is compilable in any logical context Γ. The meaning of the $\mathbf{compilable}(e)$ judgment is defined by the target architecture. A program e' is compilable if it is a sequence of valid assembly instructions. The compilation task is a process of rewriting the source program e to an equivalent assembly program e'.

2. PARSING

In order to use the formal system for program transformation, source-level programs expressed as sequences of characters must first be translated into a term representation for use in the MetaPRL framework. We assume that the source language can be specified using a context-free grammar, and traditional lexing and parsing methods can be used to perform the translation.

MetaPRL provides these capabilities using the integrated Phobos [3] generic lexer and parser, which enables users to specify parts of their logical theories using their own notation. For instance, we can use actual program notation (instead of the uniform term syntax) to express program transformations in rewrite rules and we can specify test programs in source notation.

A Phobos language specification resembles a typical parser definition in YACC [9], except that semantic actions for productions use term rewriting. Phobos employs *informal* rewriting, which means that it uses a rewriting engine that can create new variable bindings and perform capturing substitution.

In Phobos, the lexer for a language is specified as a set of lexical rewrite rules of the form $\textit{regex} \longleftrightarrow \textit{term}$, where *regex* is a special term that is created for every token and contains the matched input as a string parameter as well as a subterm containing the position in the source text, which can be used to produce more informative messages if an error is detected. The following example demonstrates a single lexer clause, that translates a nonnegative decimal number to a term with operator name `number` and a single integer parameter.

$$\texttt{NUM} \;=\; \text{"}[0-9]+\text{"}\;\; \{\mathbf{token}[i]\{\textit{pos}\} \longleftrightarrow \textit{number}[i]\}$$

The parser is defined as a set of grammar productions. For each grammar production in the program syntax shown in Figure 1, we define a production in the form

$$S ::= S_1 \ldots S_n \longleftrightarrow \textit{term}$$

where the symbols S_i may be annotated with a term pattern. For instance, the production for the let-expression is defined with the following production and semantic action.

$$\begin{array}{l}\texttt{exp} ::= \texttt{LET ID}\,\langle\texttt{v}\rangle\; \texttt{EQ exp}\,\langle\texttt{e}\rangle\; \texttt{IN exp}\,\langle\texttt{rest}\rangle \\ \longleftrightarrow \mathbf{let}\; v = e \;\mathbf{in}\; \textit{rest}\end{array}$$

Phobos constructs an LALR(1) parser from these specifications that maintains a stack of terms and applies the associated rewrite rule each time a production is reduced by replacing the corresponding terms on the stack with the result. For the parser to accept, the stack must contain a single term corresponding to the start symbol of the grammar.

It may not be feasible during parsing to create the initial binding structure of the programs. For instance, in our implementation function parameters are collected as a list and are not initially bound in the function body. Furthermore, for mutually recursive functions, the function variables are not initially bound in the functions' bodies. For this reason, the parsing phases is usually followed by an additional rewrite phase that performs these operations using the informal rewriting engine. The source text is replaced with the resulting term on completion.

3. INTERMEDIATE REPRESENTATION

The intermediate representation of the program must serve two conflicting purposes. It should be a fairly low-level language so that translation to machine code is as straightforward as possible. However, it should be abstract enough that program transformations and optimizations need not be overly concerned with implementation detail. The intermediate representation we use is similar to the functional intermediate representations used by several groups [1, 5, 18], in which the language retains a similarity to an ML-like language where all intermediate values apart from arithmetic expressions are explicitly named.

In this form, the IR is partitioned into two main parts: "atoms" define values like numbers, arithmetic, and variables; and "expressions" define all other computation. The language includes arithmetic, conditionals, tuples, functions, and function definitions, as shown in Figure 2.

Function definitions deserve special mention. Functions are defined using the $\mathbf{let\ rec}\; R = d \;\mathbf{in}\; e$ term, where d is a list of mutually recursive functions, and R represents a recursively defined record containing these functions. Each of the functions is labeled, and the term $R.l$ represents the function with label l in record R.

While this representation has an easy formal interpretation as a fixpoint of the single variable R, it is awkward to

$binop$	$::=$	$+ \mid - \mid * \mid /$	*Binary arithmetic*
$relop$	$::=$	$= \mid <> \mid \leq \mid < \mid \geq \mid >$	*Binary relations*
l	$::=$	$string$	*Function label*
a	$::=$	$\top \mid \bot$	*Boolean values*
	$\mid$	i	*Integers*
	$\mid$	v	*Variables*
	$\mid$	$a_1\ binop\ a_2$	*Binary arithmetic*
	$\mid$	$a_1\ relop\ a_2$	*Binary relations*
	$\mid$	$R.l$	*Function labels*
e	$::=$	$\mathbf{let}\ v = a\ \mathbf{in}\ e$	*Variable definition*
	$\mid$	$\mathbf{if}\ a\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2$	*Conditional*
	$\mid$	$\mathbf{let}\ v = (a_1, \ldots, a_n)\ \mathbf{in}\ e$	*Tuple allocation*
	$\mid$	$\mathbf{let}\ v = a_1[a_2]\ \mathbf{in}\ e$	*Subscripting*
	$\mid$	$a_1[a_2] \leftarrow a_3; e$	*Assignment*
	$\mid$	$\mathbf{let}\ v = a(a_1, \ldots, a_n)\ \mathbf{in}\ e$	*Function application*
	$\mid$	$\mathbf{letc}\ v = a_1(a_2)\ \mathbf{in}\ e$	*Closure creation*
	$\mid$	$\mathbf{return}\ a$	*Return a value*
	$\mid$	$a(a_1, \ldots, a_n)$	*Tail-call*
	$\mid$	$\mathbf{let\ rec}\ R = d\ \mathbf{in}\ e$	*Recursive functions*
e_λ	$::=$	$\lambda v.e_\lambda \mid \lambda v.e$	*Functions*
d	$::=$	$\mathbf{fun}\ l = e_\lambda\ \mathbf{and}\ d$	*Function definitions*
	$\mid$	ϵ	

Figure 2: Intermediate Representation

use, principally because it violates the rule of higher-order abstract syntax: namely, that (function) variables be represented as variables in the meta-language. In some sense, this representation is an artifact of the MetaPRL term language: it is not possible, given the term language described in Section 1.2, to define more than one recursive variable at a time. We are currently investigating extending the meta-language to address this problem.

3.1 AST to IR conversion

The main difference between the abstract syntax representation and the IR is that intermediate expressions in the AST do not have to be named. In addition, the conditional in the AST can be used anywhere an expression can be used (for instance, as the argument to a function), while in the IR, the branches of the conditional must be terminated by a **return** a expression or tail-call.

The translation from AST to IR is straightforward, but we use it to illustrate a style of translation we use frequently. The term $\mathtt{IR}\{e_1; v.e_2[v]\}$ (displayed as $[\![e_1]\!]_{IR} v.e_2[v]$) is the translation of an expression e_1 to an IR atom, which is substituted for v in expression $e_2[v]$. The translation problem is expressed through the following rule, which states that a program e is compilable if the program can be translated to an atom, returning the value as the result of the program.

$$\frac{\Gamma \vdash \mathbf{compilable}([\![e]\!]_{IR} v.\mathbf{return}\ v)}{\Gamma \vdash \mathbf{compilable}(e)}$$

For many AST expressions, the translation to IR is straightforward. The following rules give a few representative examples.

$$\begin{array}{ll}
[\mathrm{int}] & [\![i]\!]_{IR} v.e[v] \longleftrightarrow e[i] \\
[\mathrm{var}] & [\![v_1]\!]_{IR} v_2.e[v_2] \longleftrightarrow e[v_1] \\
[\mathrm{add}] & [\![e_1 + e_2]\!]_{IR} v.e[v] \\
\longleftrightarrow & [\![e_1]\!]_{IR} v_1.[\![e_2]\!]_{IR} v_2.e[v_1 + v_2] \\
[\mathrm{set}] & [\![e_1[e_2] \leftarrow e_3]\!]_{IR} v.e_4[v] \\
\longleftrightarrow & \quad [\![e_1]\!]_{IR} v_1. \\
 & \quad [\![e_2]\!]_{IR} v_2. \\
 & \quad [\![e_3]\!]_{IR} v_3. \\
 & \quad v_1[v_2] \leftarrow v_3; \\
 & \quad e_4[\bot]
\end{array}$$

For conditionals, code duplication is avoided by wrapping the code after the conditional in a function, and calling the function at the tail of each branch of the conditional.

$$\begin{array}{ll}
[\mathrm{if}] & [\![\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3]\!]_{IR} v.e_4[v] \\
\longleftrightarrow & \quad \mathbf{let\ rec}\ R = \mathbf{fun}\ g = \lambda v.e_4[v]\ \mathbf{and}\ \epsilon\ \mathbf{in} \\
 & \quad [\![e_1]\!]_{IR} v_1. \\
 & \quad \mathbf{if}\ v_1\ \mathbf{then}\ [\![e_2]\!]_{IR} v_2.(R.g(v_2))\ \mathbf{else}\ [\![e_3]\!]_{IR} v_3.(R.g(v_3))
\end{array}$$

For functions, the post-processing phase converts recursive function definitions to the record form, and we have the following translation, using the term $[\![d]\!]_{IR}$ to translate function definitions. In general, anonymous functions must be named *except* when they are outermost in a function definition. The post-processing phase produces two kinds of λ-abstractions, the $\lambda_p v.e[v]$ is used to label function parameters in recursive definitions, and the $\lambda v.e[v]$ term is used for anonymous functions.

$$\begin{array}{ll}
[\mathrm{letrec}] & [\![\mathbf{let\ rec}\ R = d\ \mathbf{in}\ e_1]\!]_{IR} v.e_2[v] \\
\longleftrightarrow & \mathbf{let\ rec}\ R = [\![d]\!]_{IR}\ \mathbf{in}\ [\![e_1]\!]_{IR} v.e_2[v] \\
[\mathrm{fun}] & [\![\mathbf{fun}\ l = e\ \mathbf{and}\ d]\!]_{IR} \\
\longleftrightarrow & \mathbf{fun}\ l = [\![e]\!]_{IR} v.\mathbf{return}\ v\ \mathbf{and}\ [\![d]\!]_{IR} \\
[\mathrm{param}] & [\![\lambda_p v_1.e_1[v_1]]\!]_{IR} v_2.e_2[v_2] \\
\longleftrightarrow & \lambda v_1.([\![e_1[v_1]]\!]_{IR} v_2.e_2[v_2]) \\
[\mathrm{abs}] & [\![\lambda v_1.e_1[v_1]]\!]_{IR} v_2.e_2[v_2] \\
\longleftrightarrow & \quad \mathbf{let\ rec}\ R = \\
 & \qquad \mathbf{fun}\ g = \lambda v_1.[\![e_1[v_1]]\!]_{IR} v_3.\mathbf{return}\ v_3\ \mathbf{and}\ \epsilon \\
 & \quad \mathbf{in}\ e_2[R.g]
\end{array}$$

3.2 CPS conversion

CPS conversion is an optional phase of the compiler that converts the program to continuation-passing style. That is, instead of returning a value, functions pass their results to a continuation function that is passed as an argument. In this phase, all functions become tail-calls, and all occurrences of $\mathbf{let}\ v = a_1(a_2)\ \mathbf{in}\ e$ and $\mathbf{return}\ a$ are eliminated. The main objective in CPS conversion is to pass the result of the computation to a continuation function. We state this formally as the following inference rule, which states that a program e is compilable if for all functions c, the program $[\![e]\!]_c$ is compilable.

$$\frac{\Gamma, c\colon exp \vdash \mathbf{compilable}([\![e]\!]_c)}{\Gamma \vdash \mathbf{compilable}(e)}$$

The term $[\![e]\!]_c$ represents the application of the c function to the program e, and we can use it to transform the program e by migrating the call to the continuation downward in the expression tree. Abstractly, the process proceeds as follows.

- First, replace each function definition $f = \lambda x.e[x]$ with a continuation form $f = \lambda c.\lambda x.[\![e[x]]\!]_c$ and simultaneously replace all occurrences of f with the partial application $f[\mathbf{id}]$, where **id** is the identity function.

- Next, replace tail-calls $[\![f[\mathbf{id}](a_1, \ldots, a_n)]\!]_c$ with $f(c, a_1, \ldots, a_n)$, and return statements $[\![\mathbf{return}\ a]\!]_c$ with $c(a)$.
- Finally, replace inline-calls $[\![\mathbf{let}\ v = f[\mathbf{id}](a_1, \ldots, a_n)\ \mathbf{in}\ e]\!]_c$ with the continuation-passing version $\mathbf{let\ rec}\ R = \mathbf{fun}\ g = \lambda v.[\![e]\!]_c\ \mathbf{and}\ \epsilon\ \mathbf{in}\ f(g, a_1, \ldots, a_n)$.

For many expressions, CPS conversion is a straightforward mapping of the CPS translation, as shown by the following five rules.

$$
\begin{array}{ll}
\text{[atom]} & [\![\mathbf{let}\ v = a\ \mathbf{in}\ e[v]]\!]_c \longleftrightarrow \mathbf{let}\ v = a\ \mathbf{in}\ [\![e[v]]\!]_c \\
\text{[tuple]} & [\![\mathbf{let}\ v = (a_1, \ldots, a_n)\ \mathbf{in}\ e[v]]\!]_c \\
\longleftrightarrow & \mathbf{let}\ v = (a_1, \ldots, a_n)\ \mathbf{in}\ [\![e[v]]\!]_c \\
\text{[letsub]} & [\![\mathbf{let}\ v = a_1[a_2]\ \mathbf{in}\ e[v]]\!]_c \\
\longleftrightarrow & \mathbf{let}\ v = a_1[a_2]\ \mathbf{in}\ [\![e[v]]\!]_c \\
\text{[setsub]} & [\![a_1[a_2] \leftarrow a_3; e[v]]\!]_c \longleftrightarrow a_1[a_2] \leftarrow a_3; [\![e[v]]\!]_c \\
\text{[if]} & [\![\mathbf{if}\ a\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2]\!]_c \\
\longleftrightarrow & \mathbf{if}\ a\ \mathbf{then}\ [\![e_1]\!]_c\ \mathbf{else}\ [\![e_2]\!]_c
\end{array}
$$

The modification of functions is the key part of the conversion. When a $\mathbf{let\ rec}\ R = d[R]\ \mathbf{in}\ e[R]$ term is converted, the goal is to add an extra continuation parameter to each of the functions in the recursive definition. Conversion of the function definition is shown in the *fundef* rule, where the function gets an extra continuation argument that is then applied to the function body.

In order to preserve the program semantics, we must then replace all occurrences of the function with the term $f[\mathbf{id}]$, which represents the partial application of the function to the identity. This step is performed in two parts: first the *letrec* rule replaces all occurrences of the record variable R with the term $R[\mathbf{id}]$, and then the *letfun* rule replaces each function variable f with the term $f[\mathbf{id}]$.

$$
\begin{array}{ll}
\text{[letrec]} & [\![\mathbf{let\ rec}\ R = d[R]\ \mathbf{in}\ e[R]]\!]_c \\
\longleftrightarrow & \mathbf{let\ rec}\ R = [\![d[R[\mathbf{id}]]]\!]_c\ \mathbf{in}\ [\![e[R[\mathbf{id}]]]\!]_c \\
\text{[fundef]} & [\![\mathbf{fun}\ l = \lambda v.e[v]\ \mathbf{and}\ d]\!]_c \\
\longleftrightarrow & \mathbf{fun}\ l = \lambda c.\lambda v.[\![e[v]]\!]_c\ \mathbf{and}\ [\![d]\!]_c \\
\text{[enddef]} & [\![\epsilon]\!]_c \longleftrightarrow \epsilon \\
\text{[letfun]} & [\![\mathbf{let}\ v = R[\mathbf{id}].l\ \mathbf{in}\ e[v]]\!]_c \\
\longleftrightarrow & \mathbf{let}\ v = R.l\ \mathbf{in}\ [\![e[v[\mathbf{id}]]]\!]_c
\end{array}
$$

Non-tail-call function applications must also be converted to continuation passing form, as shown in the *apply* rule, where the expression *after* the function call is wrapped in a continuation function and passed as a continuation argument.

$$
\begin{array}{ll}
\text{[apply]} & [\![\mathbf{let}\ v_2 = v_1[\mathbf{id}](a)\ \mathbf{in}\ e[v_2]]\!]_c \\
\longleftrightarrow & \mathbf{let\ rec}\ R = \mathbf{fun}\ g = \lambda v.[\![e[v]]\!]_c\ \mathbf{and}\ \epsilon\ \mathbf{in} \\
& \mathbf{let}\ g = R.g\ \mathbf{in}\ f(g; a)
\end{array}
$$

In the final phase of CPS conversion, we can replace return statements with a call to the continuation. For tail-calls, we replace the partial application of the function $f[\mathbf{id}]$ with an application to the continuation.

$$
\begin{array}{ll}
\text{[return]} & [\![\mathbf{return}\ a]\!]_c \longleftrightarrow c(a) \\
\text{[tailcall]} & [\![f[\mathbf{id}](a_1, \ldots, a_n)]\!]_c \longleftrightarrow f(c, a_1, \ldots, a_n)
\end{array}
$$

3.3 Closure conversion

The program intermediate representation includes higher-order and nested functions. The function nesting must be eliminated before code generation, and the lexical scoping of function definitions must be preserved when functions are passed as values. This phase of program translation is normally accomplished through *closure conversion*, where the free variables for nested functions are captured in an environment as passed to the function as an extra argument. The function body is modified so that references to variables that were defined outside the function are now references to the environment parameter. In addition, when a function is passed as a value, the function is paired with the environment as a *closure*.

The difficult part of closure conversion is the construction of the environment, and the modification of variables in the function bodies. We can formalize closure conversion as a sequence of steps, each of which preserves the program's semantics. In the first step, we must modify each function definition by adding a new environment parameter. To represent this, we replace each $\mathbf{let\ rec}\ R = d\ \mathbf{in}\ e$ term in the program with a new term $\mathbf{let\,rec}\, R\ \mathbf{with}\ [f = ()] = d\ \mathbf{in}\ e$, where f is an additional parameter, initialized to the empty tuple $()$, to be added to each function definition. Simultaneously, we replace every occurrence of the record variable R with $R(f)$, which represents the partial application of the record R to the tuple f.

$$
\begin{array}{ll}
\text{[frame]} & \mathbf{let\ rec}\ R = d[R]\ \mathbf{in}\ e[R] \\
\longleftrightarrow & \mathbf{let\,rec}\, R\ \mathbf{with}\ [f = ()] = d[R(f)]\ \mathbf{in}\ e[R(f)]
\end{array}
$$

The second part of closure conversion does the closure operation using two operations. For the first part, suppose we have some expression e with a free variable v. We can abstract this variable using a call-by-name function application as the expression $\mathbf{let}\ v = v\ \mathbf{in}\ e$, which reduces to e by simple β-reduction.

$$
\text{[abs]} \qquad e[v] \longleftrightarrow \mathbf{let}\ v = v\ \mathbf{in}\ e[v]
$$

By selectively applying rule, we can quantify variables that occur free in the function definitions d in a term $\mathbf{let\,rec}\, R\ \mathbf{with}\ [f = \mathit{tuple}] = d\ \mathbf{in}\ e$. The main closure operation is the addition of the abstracted variable to the frame, using the following rewrite.

$$
\begin{array}{ll}
\text{[close]} & \mathbf{let}\ v = a\ \mathbf{in} \\
& \mathbf{let\,rec}\, R\ \mathbf{with}\ [f = (a_1, \ldots, a_n)] = \\
& \quad d[R; v; f] \\
& \mathbf{in}\ e[R; v; f] \\
\longleftrightarrow & \mathbf{let\,rec}\, R\ \mathbf{with}\ [f = (a_1, \ldots, a_n, a)] = \\
& \quad \mathbf{let}\ v = f[n+1]\ \mathbf{in}\ d[R; v; f] \\
& \mathbf{in\ let}\ v = a\ \mathbf{in}\ e[R; v; f]
\end{array}
$$

Once all free variables have been added to the frame, the $\mathbf{let\,rec}\, R\ \mathbf{with}\ [f = \mathit{tuple}] = d\ \mathbf{in}\ e$ rewritten to use explicit tuple allocation.

$$
\begin{array}{ll}
\text{[alloc]} & \textbf{let rec } R \textbf{ with } [f = \mathit{tuple}] = \\
& \quad d[R; f] \\
& \textbf{in } e[R; f] \\
\longleftrightarrow & \textbf{let rec } R = \textbf{frame}(f, d[R; f]) \textbf{ in} \\
& \textbf{let } f = (\mathit{tuple}) \textbf{ in } e[R; f]
\end{array}
$$

The final step of closure conversion is to propagate the subscript operations into the function bodies.

$$
\begin{array}{ll}
\text{[arg]} & \textbf{frame}(f, \textbf{fun } l = \lambda v.e[f; v] \textbf{ and } d[f]) \\
\longleftrightarrow & \textbf{fun } l = \lambda f.\lambda v.e[f; v] \textbf{ and frame}(f, d[\mathit{frame}]) \\
\text{[sub]} & \textbf{let } v_1 = a_1[a_2] \textbf{ in fun } l = \lambda v_2.e[v_1; v_2] \textbf{ and } d[v_1] \\
\longleftrightarrow & \quad \textbf{fun } l = \lambda v_2. \textbf{let } v_1 = a_1[a_2] \textbf{ in } e[v_1; v_2] \textbf{ and} \\
& \quad \textbf{let } v_1 = a_1[a_2] \textbf{ in } d[v_1]
\end{array}
$$

3.4 IR optimizations

Many optimizations on the intermediate representation are quite easy to express. For illustration, we include two very simple optimizations: dead-code elimination and constant folding.

3.4.1 Dead code elimination

Formally, an expression e in a program p is dead if the removal of expression e does not change the behavior of the program p. Complete elimination of dead-code is undecidable: for example, an expression e is dead if no program execution ever reaches expression e. The most frequent approximation is based on scoping: a let-expression $\textbf{let } v = a \textbf{ in } e$ is dead if v is not free in e. This kind of dead-code elimination can be specified with the following set of rewrites.

$$
\begin{array}{ll}
\text{[datom]} & \textbf{let } v = a \textbf{ in } e \longleftrightarrow e \\
\text{[dtuple]} & \textbf{let } v = (a_1, \ldots, a_n) \textbf{ in } e \longleftrightarrow e \\
\text{[dsub]} & \textbf{let } v = a_1[a_2] \textbf{ in } e \longleftrightarrow e \\
\text{[dcl]} & \textbf{letc } v = a_1(a_2) \textbf{ in } e \longleftrightarrow e
\end{array}
$$

The syntax of these rewrites depends on the second-order specification of substitution. Note that the pattern e is *not* expressed as the second-order pattern $e[v]$. That is, v is *not* allowed to occur free in e.

Furthermore, note that dead-code elimination of this form is aggressive. For example, suppose we have an expression $\textbf{let } v = a \ / \ 0 \textbf{ in } e$. This expression is considered as dead-code even though division by 0 is not a valid operation. If the target architecture raises an exception on division by zero, this kind of aggressive dead-code elimination is unsound. This problem can be addressed formally by partitioning the class of atoms into two parts: those that may raise an exception, and those that do not, and applying dead-code elimination only on the first class. The rules for dead-code elimination are the same as above, where the calls of atom a refers only to those atoms that do not raise exceptions.

3.4.2 Constant-folding

Another simple class of optimizations is constant folding. If we have an expression that includes only constant values, the expression may be computed at compile time. The following rewrite captures the arithmetic part of this optimization, where $[\![op]\!]$ is the interpretation of the arithmetic operator in the meta-language. Relations and conditionals can be folded in a similar fashion.

$$
\begin{array}{ll}
\text{[binop]} & i \ \mathit{binop} \ j \longleftrightarrow [\![op]\!](i, j) \\
\text{[relop]} & i \ \mathit{relop} \ j \longleftrightarrow [\![op]\!](i, j) \\
\text{[ift]} & \textbf{if } \top \textbf{ then } e_1 \textbf{ else } e_2 \longleftrightarrow e_1 \\
\text{[iff]} & \textbf{if } \bot \textbf{ then } e_1 \textbf{ else } e_2 \longleftrightarrow e_2
\end{array}
$$

In order for these transformations to be faithful, the arithmetic must be performed over the numeric set provided by the target architecture (our implementation, described in Section 4.2, uses 31-bit signed integers).

For simple constants a, it is usually more efficient to inline the $\textbf{let } v = a \textbf{ in } e[v]$ expression as well.

$$
\begin{array}{ll}
\text{[cint]} & \textbf{let } v = i \textbf{ in } e[v] \longleftrightarrow e[i] \\
\text{[cfalse]} & \textbf{let } v = \bot \textbf{ in } e[v] \longleftrightarrow e[\bot] \\
\text{[ctrue]} & \textbf{let } v = \top \textbf{ in } e[v] \longleftrightarrow e[\top] \\
\text{[cvar]} & \textbf{let } v_2 = v_1 \textbf{ in } e[v_2] \longleftrightarrow e[v_1]
\end{array}
$$

4. SCOPED X86 ASSEMBLY LANGUAGE

Once closure conversion has been performed, all function definitions are top-level and closed, and it becomes possible to generate assembly code. When formalizing the assembly code, we continue to use higher-order abstract syntax: registers and variables in the assembly code correspond to variables in the meta-language. There are two important properties we must maintain. First, scoping must be preserved: there must be a binding occurrence for each variable that is used. Second, in order to facilitate reasoning about the code, variables/registers must be immutable.

These two requirements seem at odds with the traditional view of assembly, where assembly instructions operate by side-effect on a finite register set. In addition, the Intel x86 instruction set architecture primarily uses two-operand instructions, where the value in one operand is both used and modified in the same instruction. For example, the instruction $\mathit{ADD}\ r_1,r_2$ performs the operation $r_1 \leftarrow r_1 + r_2$, where r_1 and r_2 are registers.

To address these issues, we define an abstract version of the assembly language that uses a three operand version on the instruction set. The instruction $\mathit{ADD}\ v_1,\ v_2,\ \lambda v_3.e$ performs the abstract operation $\textbf{let } v_3 = v_1 + v_2 \textbf{ in } e$. The variable v_3 is a *binding* occurrence, and it is bound in body of the instruction e. In our account of the instruction set, *every* instruction that modifies a register has a binding occurrence of the variable being modified. Instructions that *do not* modify memory use the traditional non-binding form of the instruction. For example, the instruction $\mathit{ADD}\ v_1,\ (\%v_2); e$ performs the operation $(\%v_2) \leftarrow (\%v_2) + v_1$, where $(\%v_2)$ means the value in memory at location v_2.

The complete abstract instruction set that we use is shown in Figure 3 (the Intel x86 architecture includes a large number of complex instructions that we do not use). Instructions may use several forms of operands and addressing modes.

- The *immediate* operand $\$i$ is a constant number i.
- The *label* operand $\$R.l$ refers to the address of the function in record R labeled l.
- The *register* operand $\%v$ refers to register/variable v.
- The *indirect* operand $(\%v)$ refers to the value in memory at location v.

l	$::=$	string	Function labels
r	$::=$	$eax \mid ebx \mid ecx \mid edx$	Registers
	$\mid$	$esi \mid edi \mid esp \mid ebp$	
v	$::=$	$r \mid v_1, v_2, \ldots$	Variables
o_m	$::=$	$(\%v)$	Memory operands
	$\mid$	$i(\%v)$	
	$\mid$	$i_1(\%v_1, \%v_2, i_2)$	
o_r	$::=$	$\%v$	Register operand
o	$::=$	$o_m \mid o_r$	General operands
	$\mid$	$\$i$	Constant number
	$\mid$	$\$v.l$	Label
cc	$::=$	$= \mid <> \mid < \mid > \mid \leq \mid \geq$	Condition codes
$\mathit{inst1}$	$::=$	$\mathit{INC} \mid \mathit{DEC} \mid \cdots$	1-operand opcodes
$\mathit{inst2}$	$::=$	$\mathit{ADD} \mid \mathit{SUB} \mid \mathit{AND} \mid \cdots$	2-operand opcodes
$\mathit{inst3}$	$::=$	$\mathit{MUL} \mid \mathit{DIV}$	3-operand opcodes
cmp	$::=$	$\mathit{CMP} \mid \mathit{TEST}$	comparisons
jmp	$::=$	JMP	unconditional branch
jcc	$::=$	$\mathit{JEQ} \mid \mathit{JLT} \mid \mathit{JGT} \mid \cdots$	conditional branch
e	$::=$	$\mathit{MOV}\ o,\ \lambda v.e$	Copy
	$\mid$	$\mathit{inst1}\ o_m; e$	1-operand mem inst
	$\mid$	$\mathit{inst1}\ o_r,\ \lambda v.e$	1-operand reg inst
	$\mid$	$\mathit{inst2}\ o_r,\ o_m; e$	2-operand mem inst
	$\mid$	$\mathit{inst2}\ o,\ o_r,\ \lambda v.e$	2-operand reg inst
	$\mid$	$\mathit{inst3}\ o,\ o_r,\ o_r,\ \lambda v_1, v_2.e$	3-operand reg inst
	$\mid$	$\mathit{cmp}\ o_1,\ o_2$	Comparison
	$\mid$	$\mathit{jmp}\ o(o_r; \ldots; o_r)$	Unconditional branch
	$\mid$	$j\,cc\ \textbf{then}\ e_1\ \textbf{else}\ e_2$	Conditional branch
p	$\mid$	$\textbf{let rec}\ R = d\ \textbf{in}\ p \mid e$	Programs
d	$\mid$	$l = e_\lambda\ \textbf{and}\ d \mid \epsilon$	Function definition
e_λ	$::=$	$\lambda v.e_\lambda \mid e$	Functions

Figure 3: Scoped Intel x86 instruction set

- The *indirect offset* operand $i(\%v)$ refers to the value in memory at location $v + i$.
- The *array indexing* operand $i_1(\%v_1, \%v_2, i_2)$ refers to the value in memory at location $v_1 + v_2 * i_2 + i_1$, where $i_2 \in \{1, 2, 4, 8\}$.

The instructions can be placed in several main categories.

- MOV instructions copy a value from one location to another. The instruction $\mathit{MOV}\ o_1,\ \lambda v_2.e[v_2]$ copies the value in operand o_1 to variable v_2.
- One-operand instructions have the forms $\mathit{inst1}\ o_1; e$ (where o_1 must be an indirect operand), and $\mathit{inst1}\ v_1,\ \lambda v_2.e$. For example, the instruction $\mathit{INC}\ (\%r_1); e$ performs the operation $(\%r_1) \leftarrow (\%r_1) + 1; e$; and the instruction $\mathit{INC}\ \%r_1,\ \lambda r_2.e$ performs the operation $\textbf{let}\ r_2 = r_1 + 1\ \textbf{in}\ e$.
- Two-operand instructions have the forms $\mathit{inst2}\ o_1, o_2; e$, where o_2 must be an indirect operand; and $\mathit{inst2}\ o_1,\ v_2,\ \lambda v_3.e$. For example, the instruction $\mathit{ADD}\ \%r_1,\ (\%r_2); e$ performs the operation $(\%r_2) \leftarrow (\%r_2) + r_1; e$; and the instruction $\mathit{ADD}\ o_1,\ v_2,\ \lambda v_3.e$ is equivalent to $\textbf{let}\ v_3 = o_1 + v_2\ \textbf{in}\ e$.
- There are two three-operand instructions: one for multiplication and one for division, having the form $\mathit{inst3}\ o_1,\ v_2,\ v_3,\ \lambda v_4, v_5.e$. For example, the instruction $\mathit{DIV}\ \%r_1,\ \%r_2,\ \%r_3,\ \lambda r_4, r_5.e$ performs the following operation, where (r_2, r_3) is the 64-bit value $r_2 * 2^{32} + r_3$. The Intel specification requires that r_4 be the register eax, and r_5 the register edx.

$$\begin{array}{l} \textbf{let}\ r_4 = (r_2, r_3)/r_1\ \textbf{in} \\ \textbf{let}\ r_5 = (r_2, r_3)\ \textbf{mod}\ r_1\ \textbf{in} \\ \quad e \end{array}$$

- The comparison operand has the form $\mathit{CMP}\ o_1,\ o_2; e$, where the processor's condition code register is modified by the instruction. We do not model the condition code register explicitly in our current account. However, doing so would allow more greater flexibility during code-motion optimizations on the assembly.
- The unconditional branch operation $\mathit{JMP}\ o(o_1, \ldots, o_n)$ branches to the function specified by operand o, with arguments $(o_1, \ldots, o_n)$. The arguments are provided so that the calling convention may be enforced.
- The conditional branch operation $J\,cc\ \textbf{then}\ e_1\ \textbf{else}\ e_2$ is a conditional. If the condition-code matches the value in the processor's condition-code register, then the instruction branches to expression e_1; otherwise it branches to expression e_2.
- Functions are defined using the $\textbf{let rec}\ R = d\ \textbf{in}\ e$ which corresponds exactly to the same expression in the intermediate representation. The subterm d is a list of function definitions, and e is an assembly program. Functions are defined with the $\lambda v.e$, where v is a function parameter in instruction sequence e.

4.1 Translation to concrete assembly

Since the instruction set as defined is abstract, and contains binding structure, it must be translated before actual generation of machine code. The first step in doing this is register allocation: every variable in the assembly program must be assigned to an actual machine register. This step corresponds to an α-conversion where variables are renamed to be the names of actual registers; the formal system merely validates the renaming. We describe this phase in the section on register allocation 4.3.

The final step is to generate the actual program from the abstract program. This requires only local modifications, and is implemented during printing of the program (that is, it is implemented when the program is exported to an external assembler). The main translation is as follows.

- Memory instructions $\mathit{inst1}\ o_m; e$, $\mathit{inst2}\ o_r,\ o_m; e$, and $\mathit{cmp}\ o_1,\ o_2; e$ can be printed directly.
- Register instructions with binding occurrences require a possible additional mov instruction. For the 1-operand instruction $\mathit{inst1}\ o_r,\ \lambda r.e$, if $o_r = \%r$, then the instruction is implemented as $\mathit{inst1}\quad r$. Otherwise, it is implemented as the two-instruction sequence:

$$\begin{array}{ll} \text{MOV} & o_r, \%r \\ \text{inst1} & \%r \end{array}$$

Similarly, the two-operand instruction *inst2* o, o_r, $\lambda r.e$ may require an addition `mov` from o_r to r, and the three-operand instruction *inst3* o, o_{r1}, o_{r2}, $\lambda r_1, r_2.e$ may require two additional `mov` instructions.

- The $JMP\ o(o_1, \ldots, o_n)$ prints as `JMP o`. This assumes that the calling convention has been satisfied during register allocation, and all the arguments are in the appropriate places.

- The Jcc **then** e_1 **else** e_2 instruction prints as the following sequence, where cc' is the inverse of cc, and l is a new label.

$$
\begin{array}{ll}
\mathrm{J}cc' & \mathrm{l} \\
 & e_1 \\
\mathrm{l:} & e_2
\end{array}
$$

- A function definition $l = e$ **and** d in a record **let rec** $R = d$ **in** e is implemented as a labeled assembly expression $R.l\colon e$. We assume that the calling convention has been established, and the function abstraction $\lambda v.e$ ignores the parameter v, assembling only the program e.

The compiler back-end then has three stages: 1) code generation, 2) register allocation, and 3) peephole optimization, described in the following sections.

4.2 Assembly code generation

The production of assembly code is primarily a straightforward translation of operations in the intermediate code to operations in the assembly. There are two main kinds of translations: translations from atoms to operands, and translation of expressions into instruction sequences. We express these translations with the term $[\![e]\!]_{\mathbf{a}}$, which is the translation of the IR expression e to an assembly expression; and $[\![a]\!]_{\mathbf{a}} v.e$, which produces the assembly operand for the atom a and substitutes it for the variable v in expression e.

4.2.1 Atom translation

The translation of atoms is primarily a translation of the IR names for values and the assembly names for operands. A representative set of atom translations is shown in Figure 4. Since the language is untyped, we use a 31-bit representation of integers, where the least-significant-bit is always set to 1. Since pointers are always word-aligned, this allows the garbage collector to differentiate between integers and pointers. The division operation is the most complicated translation: first the operands a_1 and a_2 are shifted to obtain the standard integer representation, the division operation is performed, and the result is converted to a 31-bit representation.

4.2.2 Expression translation

Expressions translate to sequences of assembly instructions. A representative set of translations in shown in Figure 5. The translation of **let** $v = a$ **in** $e[v]$ is the simplest case, the atom a is translated into an operand v', which is copied to a variable v (since the expression $e[v]$ assumes v is a variable), and the rest of the code $e[v]$ is translated. Conditionals translate into comparisons followed by a conditional branch.

$$
\begin{array}{ll}
\text{[false]} & [\![\bot]\!]_{\mathbf{a}} v.e[v] \longleftrightarrow e[\$1] \\
\text{[true]} & [\![\top]\!]_{\mathbf{a}} v.e[v] \longleftrightarrow e[\$3] \\
\text{[int]} & [\![i]\!]_{\mathbf{a}} v.e[v] \longleftrightarrow e[\$i * 2 + 1] \\
\text{[var]} & [\![v_1]\!]_{\mathbf{a}} v_2.e[v_2] \longleftrightarrow e[\%v] \\
\text{[label]} & [\![R.l]\!]_{\mathbf{a}} v.e[v] \longleftrightarrow e[\$R.l] \\
\text{[add]} & [\![a_1 + a_2]\!]_{\mathbf{a}} v.e[v] \\
\longleftrightarrow & \quad [\![a_1]\!]_{\mathbf{a}} v_1. \\
 & \quad [\![a_2]\!]_{\mathbf{a}} v_2. \\
 & \quad \mathit{ADD}\ v_2,\ v_1,\ \lambda \mathit{tmp}. \\
 & \quad \mathit{DEC}\ \%\mathit{tmp},\ \lambda \mathit{sum}. \\
 & \quad e[\%\mathit{sum}] \\
\text{[div]} & [\![a_1\ /\ a_2]\!]_{\mathbf{a}} v.e[v] \\
\longleftrightarrow & \quad [\![a_1]\!]_{\mathbf{a}} v_1. \\
 & \quad [\![a_2]\!]_{\mathbf{a}} v_2. \\
 & \quad \mathit{SAR}\ \$1,\ v_1,\ \lambda v_1'. \\
 & \quad \mathit{SAR}\ \$1,\ v_2,\ \lambda v_2'. \\
 & \quad \mathit{MOV}\ \$0,\ \lambda v_3. \\
 & \quad \mathit{DIV}\ \%v_1',\ \%v_2',\ \%v_3',\ \lambda q', r'. \\
 & \quad \mathit{SHL}\ \$1,\ \%q',\ \lambda q''. \\
 & \quad \mathit{OR}\ \$1,\ \%q'',\ \lambda q. \\
 & \quad e[\%q]
\end{array}
$$

Figure 4: Translation of atoms to x86 assembly

The memory operations shown in Figure 6 are among the most complicated translations. For the runtime, we use a contiguous heap and a copying garbage collector. The representation of all memory blocks in the heap includes a header word containing the number of bytes in the block (the number of bytes is always a multiple of the word size), following by one word for each field. A pointer to a block points to the first field of the block (the word after the header word). The heap area itself is contiguous, delimited by *base* and *limit* pointers; the next allocation point is in the *next* pointer. These pointers are accessed through the **context**[*name*] pseudo-operand, which is later translated to an absolute memory address.

During a subscript operation, shown in the `sub` translation, the index is compared against the number of words in the block as indicated in the header word, and a bounds-check exception is raised if the index is out-of-bounds (denoted with the instruction JAE **then** *bounds.error* **else**). When a block of memory is allocated in the `alloc` and `closure` rules, the first step reserves storage with the **reserve**(i) term, and then the data is allocated and initialized. Figure 7 shows the implementation of some of the helper terms: the **reserve**(i) expression determines whether sufficient storage is present for an allocation of i bytes, and calls the garbage collector otherwise; the **store_tuple**(p, i, args); e term generates the code to initialize the fields of a tuple from a set of arguments; and the **copy_args**($\mathit{args}, \mathit{vargs}$)$\lambda v.e$ term copies the argument list in *args* into registers.

4.3 Register allocation

Register allocation is one of the easier phases of the compiler formally: the main objective of register allocation is to rename the variables in the program to use register names. The formal problem is just an α-conversion, which can be checked readily by the formal system. From a practical standpoint, however, register allocation is a NP-complete

$$
\begin{array}{ll}
\text{[atom]} & \llbracket\, \textbf{let}\ v = a\ \textbf{in}\ e[v] \rrbracket_{\mathbf{a}} \\
\longleftrightarrow & \quad \llbracket a \rrbracket_{\mathbf{a}} v'. \\
 & \quad \mathit{MOV}\ v',\ \lambda v. \\
 & \quad \llbracket e[v] \rrbracket_{\mathbf{a}} \\
\text{[if1]} & \llbracket\, \textbf{if}\ a\ \textbf{then}\ e_1\ \textbf{else}\ e_2 \rrbracket_{\mathbf{a}} \\
\longleftrightarrow & \quad \llbracket a \rrbracket_{\mathbf{a}} \mathit{test}. \\
 & \quad \mathit{CMP}\ \$0,\ \mathit{test} \\
 & \quad J\llbracket e_1 \rrbracket_{\mathbf{a}}\ \textbf{then}\ \llbracket e_2 \rrbracket_{\mathbf{a}}\ \textbf{else} \\
\text{[if2]} & \llbracket\, \textbf{if}\ a_1\ \mathit{op}\ a_2\ \textbf{then}\ e_1\ \textbf{else}\ e_2 \rrbracket_{\mathbf{a}} \\
\longleftrightarrow & \quad \llbracket a_1 \rrbracket_{\mathbf{a}} v_1. \\
 & \quad \llbracket a_2 \rrbracket_{\mathbf{a}} v_2. \\
 & \quad \mathit{CMP}\ v_1,\ v_2 \\
 & \quad J\llbracket \mathit{op} \rrbracket_{\mathbf{a}}\ \textbf{then}\ \llbracket e_1 \rrbracket_{\mathbf{a}}\ \textbf{else}\ \llbracket e_2 \rrbracket_{\mathbf{a}} \\
\text{[sub]} & \llbracket\, \textbf{let}\ v =\ a_1[a_2]\ \textbf{in}\ e[v] \rrbracket_{\mathbf{a}} \\
\longleftrightarrow & \quad \llbracket a_1 \rrbracket_{\mathbf{a}} v_1. \\
 & \quad \llbracket a_2 \rrbracket_{\mathbf{a}} v_2. \\
 & \quad \mathit{MOV}\ v_1,\ \lambda \mathit{tuple}. \\
 & \quad \mathit{MOV}\ v_2,\ \lambda \mathit{index}'. \\
 & \quad \mathit{SAR}\ \$1,\ \%\mathit{index}',\ \lambda \mathit{index}. \\
 & \quad \mathit{MOV}\ \ -4(\%\mathit{tuple}),\ \lambda \mathit{size}'. \\
 & \quad \mathit{SAR}\ \$2,\ \%\mathit{size}',\ \lambda \mathit{size}. \\
 & \quad \mathit{CMP}\ \mathit{size},\ \mathit{index} \\
 & \quad \mathit{JAE}\ \textbf{then}\ \mathit{bounds.error}\ \textbf{else} \\
 & \quad \mathit{MOV}\ 0(\%\mathit{tuple}, \%\mathit{index}, 4),\ \lambda v. \\
 & \quad \llbracket e[v] \rrbracket_{\mathbf{a}}
\end{array}
$$

Figure 5: Translation of expressions to x86 assembly

$$
\begin{array}{ll}
\text{[alloc]} & \llbracket\, \textbf{let}\ v =\ (\mathit{tuple})\ \textbf{in}\ e[v] \rrbracket_{\mathbf{a}} \\
\longleftrightarrow & \quad \textbf{reserve}(\$ \mid \mathit{tuple} \mid) \\
 & \quad \mathit{MOV}\ \textbf{context}[\mathit{next}],\ \lambda v. \\
 & \quad \mathit{ADD}\ \$(\mid \mathit{tuple} \mid +1) * 4,\ \textbf{context}[\mathit{next}] \\
 & \quad \mathit{MOV}\ \$ \mid \mathit{tuple} \mid *4,\ (\%v) \\
 & \quad \mathit{ADD}\ \$4,\ \%v,\ \lambda p. \\
 & \quad \textbf{store\_tuple}(p, 0, \mathit{tuple}); \\
 & \quad \llbracket e[v] \rrbracket_{\mathbf{a}} \\
\text{[closure]} & \llbracket\, \textbf{letc}\ v = a_1(a_2)\ \textbf{in}\ e[v] \rrbracket_{\mathbf{a}} \\
\longleftrightarrow & \quad \textbf{reserve}(\$3) \\
 & \quad \mathit{MOV}\ \textbf{context}[\mathit{next}],\ \lambda v. \\
 & \quad \mathit{ADD}\ \$12,\ \textbf{context}[\mathit{next}] \\
 & \quad \mathit{MOV}\ \$8,\ (\%v) \\
 & \quad \llbracket a_1 \rrbracket_{\mathbf{a}} v_1. \\
 & \quad \llbracket a_2 \rrbracket_{\mathbf{a}} v_2. \\
 & \quad \mathit{MOV}\ v_1,\ 4(\%v) \\
 & \quad \mathit{MOV}\ v_2,\ 8(\%v) \\
 & \quad \mathit{ADD}\ \$4,\ \%v,\ \lambda p. \\
 & \quad \llbracket e[p] \rrbracket_{\mathbf{a}} \\
\text{[call]} & \llbracket' a(\mathit{args}) \rrbracket_{\mathbf{a}} \\
\longleftrightarrow & \quad \llbracket a \rrbracket_{\mathbf{a}} \mathit{closure}. \\
 & \quad \mathit{MOV}\ 4(\%\mathit{closure}),\ \lambda \mathit{env}. \\
 & \quad \textbf{copy\_args}((),\ \mathit{args}) \lambda \mathit{vargs}. \\
 & \quad \mathit{JMP}\ (\%\mathit{closure})(\mathit{vargs})
\end{array}
$$

Figure 6: Translation of memory operations to x86 assembly

problem, and the majority of the code in our implementation is devoted to a Chaitin-style [2] graph-coloring register allocator. These kinds of allocators have been well-studied, and we do not discuss the details of the allocator here. The overall structure of the register allocator algorithm is as follows.

1. Given a program p, run a register allocator $R(p)$.
2. If the register allocator $R(p)$ was successful, it returns an assignment of variables to register names; α-convert the program using this variable assignment, and return the result p'.
3. Otherwise, if the register allocator $R(p)$ was not successful, it returns a set of variables to "spill" into memory. Rewrite the program to add fetch/store code for the spilled registers, generating a new program p', and run register allocation $R(p')$ on the new program.

Part 2 is a trivial formal operation (the logical framework checks that $p' = p$). The generation of spill code for part 3 is not trivial however, as we discuss in the following section.

4.4 Generation of spill code

The generation of spill code can affect the performance of a program dramatically, and it is important to minimize the amount of memory traffic. Suppose the register allocator was not able to generate a register assignment for a program p, and instead it determines that variable v must be placed in memory. We can allocate a new global variable, say spill_i for this purpose, and replace all occurrences of the variable with a reference to the new memory location. This can be captured by rewriting the program just after the binding occurrences of the variables to be spilled. The following two rules give an example.

$$
\begin{array}{ll}
\text{[smov]} & \mathit{MOV}\ o,\ \lambda v.e[v] \longleftrightarrow \mathit{MOV}\ o,\ \lambda \mathit{spill}_i.e[\mathit{spill}_i] \\
\text{[sinst2]} & \mathit{inst2}\ o,\ o_r,\ \lambda v.e[v] \\
\longleftrightarrow & \quad \mathit{MOV}\ o_r,\ \lambda \mathit{spill}_i. \\
 & \quad \mathit{inst2}\ o,\ \mathit{spill}_i \\
 & \quad e[\mathit{spill}_i]
\end{array}
$$

However, this kind of brute-force approach spills *all* of the occurrences of the variable, even those occurrences that could have been assigned to a register. Furthermore, the spill location spill_i would presumably be represented as the label of a memory location, not a variable, allowing a conflicting assignment of another variable to the same spill location.

To address both of these concerns, we treat spill locations as variables, and introduce scoping for spill variables. We introduce two new pseudo-operands, and two new instructions, shown in Figure 8. The instruction $\mathit{SPILL}\ o_r,\ \lambda s.e[s]$ generates a new spill location represented in the variable s, and stores the operand o_r in that spill location. The operand $\textbf{spill}[v,\ s]$ represents the value in spill location s, and it also specifies that the values in spill location s and in the register v are the same. The operand $\textbf{spill}[s]$ refers the the value in spill location s. The value in a spill operand is retrieved with the $\mathit{SPILL}\ o_s,\ \lambda v.e[v]$ and placed in the variable v.

The actual generation of spill code then proceeds in two main phases. Given a variable to spill, the first phase generates the code to store the value in a new spill location, then adds copy instruction to split the live range of the variable so that all uses of the variable refer to different freshly-generated operands of the form $\textbf{spill}[v,\ s]$. For ex-

$$
\begin{array}{ll}
\text{[reserve]} & \mathbf{reserve}(i); e \\
\longleftrightarrow & \quad \mathit{MOV}\ \mathbf{context}[\mathit{limit}],\ \lambda \mathit{limit}. \\
& \quad \mathit{SUB}\ \mathbf{context}[\mathit{next}],\ \%\mathit{limit},\ \lambda \mathit{free}. \\
& \quad \mathit{CMP}\ i,\ \%\mathit{free} \\
& \quad J\mathbf{b}\ \mathbf{then}\ gc(i)\ \mathbf{else}\ e \\
\text{[stuple1]} & \mathbf{store\_tuple}(p, i, (a :: \mathit{args})); e \\
\longleftrightarrow & \quad [\![a]\!]_{\mathbf{a}} v. \\
& \quad \mathit{MOV}\ v,\ i(\%p) \\
& \quad \mathbf{store\_tuple}(p, i+4, \mathit{args}); e \\
\text{[stuple2]} & \mathbf{store\_tuple}(p, i, ()); e \longleftrightarrow e \\
\text{[copy1]} & \mathbf{copy\_args}((a :: \mathit{args}), \mathit{vargs}) \lambda v.e[v] \\
\longleftrightarrow & \quad [\![a]\!]_{\mathbf{a}} v'. \\
& \quad \mathit{MOV}\ v',\ \lambda v. \\
& \quad \mathbf{copy\_args}(\mathit{args}, (\%v :: \mathit{vargs})) \lambda v.e[v] \\
\text{[copy2]} & \mathbf{copy\_args}((), \mathit{vargs}) \lambda v.e[v] \longleftrightarrow e[\mathbf{reverse}(\mathit{vargs})]
\end{array}
$$

Figure 7: Auxiliary terms for x86 code generation

$$
\begin{array}{rcll}
o_s & ::= & \mathbf{spill}[v,\ s] & \text{Spill operands} \\
& | & \mathbf{spill}[s] & \\
\\
e & ::= & \mathit{SPILL}\ o_r,\ \lambda s.e[s] & \text{New spill} \\
& | & \mathit{SPILL}\ o_s,\ \lambda v.e[v] & \text{Get the spilled value}
\end{array}
$$

Figure 8: Spill pseudo-operands and instructions

ample, consider the following code fragment, and suppose the register allocator determines that the variable v is to be spilled, because a register cannot be assigned in code segment 2.

$$
\begin{array}{l}
\mathit{AND}\ o,\ o_r,\ \lambda v. \\
\text{...code segment 1...} \\
\mathit{ADD}\ \%v,\ o \\
\text{...code segment 2...} \\
\mathit{SUB}\ \%v,\ o \\
\text{...code segment 3...} \\
\mathit{OR}\ \%v,\ o
\end{array}
$$

The first phase rewrites the code as follows. The initial occurrence of the variable is spilled into a new spill location s. The value is fetched just before each use of the variable, and copied to a new register. Note that the later uses refer to the new registers, creating a copying daisy-chain, but the registers have not been actually eliminated.

$$
\begin{array}{l}
\mathit{AND}\ o,\ o_r,\ \lambda v_1. \\
\mathit{SPILL}\ \%v_1,\ \lambda s. \\
\text{...code segment 1...} \\
\mathit{SPILL}\ \mathbf{spill}[v_1,\ s],\ \lambda v_2. \\
\mathit{ADD}\ \%v_2,\ o \\
\text{...code segment 2...} \\
\mathit{SPILL}\ \mathbf{spill}[v_2,\ s],\ \lambda v_3. \\
\mathit{SUB}\ \%v_3,\ o \\
\text{...code segment 3...} \\
\mathit{SPILL}\ \mathbf{spill}[v_3,\ s],\ \lambda v_4. \\
\mathit{OR}\ \%v,\ o
\end{array}
$$

Once the live range is split, the register allocator has the freedom to spill only part of the live range. During the second phase of spilling, the allocator will determine that register v_2 must be spilled in code segment 2, and the **spill**$[v_2,\ s]$ operand is replaced with **spill**$[s]$ forcing the fetch from memory, not the register v_2. Register v_2 is no longer live in code segment 2, easing the allocation task without also spilling the register in code segments 1 and 3.

4.5 Formalizing spill code generation

The formalization of spill code generation can be performed in three parts. The first part generates new spill locations (line 2 in the code sequence above); the second part generates live-range splitting code (lines 4, 7, and 10); and the third part replaces operands of the form **spill**$[v,\ s]$ with **spill**$[s]$ when requested by the garbage collector.

The first part requires a rewrite for each kind of instruction that contains a binding occurrence of a variable. The following two rewrites are representative examples. Note that all occurrences of the variable v are replaced with **spill**$[v,\ s]$, potentially generating operands like $i(\%\mathbf{spill}[v,\ s])$. These kinds of operands are rewritten at the end of spill-code generation to their original form, e.g. $i(\%v)$.

$$
\begin{array}{ll}
\text{[smov]} & \mathit{MOV}\ o_r,\ \lambda v.e[v] \\
\longleftrightarrow & \quad \mathit{MOV}\ o_r,\ \lambda v. \\
& \quad \mathit{SPILL}\ \%v,\ \lambda s. \\
& \quad e[\mathbf{spill}[v,\ s]] \\
\text{[sinst2]} & \mathit{inst2}\ o,\ o_r,\ \lambda v.e[v] \\
\longleftrightarrow & \quad \mathit{inst2}\ o,\ o_r,\ \lambda v.e[v] \\
& \quad \mathit{SPILL}\ \%v,\ \lambda s. \\
& \quad e[\mathbf{spill}[v,\ s]]
\end{array}
$$

The second rewrite splits a live range of a spill at an arbitrary point. This rewrite applies to any program that contains an occurrence of an operand **spill**$[v_1,\ s]$, and translates it to a new program that fetches the spill into a new register v_2 and uses the new spill operand **spill**$[v_2,\ s]$ in the remainder of the program. This rewrite is selectively applied before any instruction that uses an operand **spill**$[v_1,\ s]$.

$$
\begin{array}{ll}
\text{[split]} & e[\mathbf{spill}[v_1,\ s]] \\
\longleftrightarrow & \quad \mathit{SPILL}\ \mathbf{spill}[v_1,\ s],\ \lambda v_2.e[\mathbf{spill}[v_2,\ s]]
\end{array}
$$

In the third and final phase, when the register allocator determines that a variable should be spilled, the **spill**$[v,\ s]$ operands are selectively eliminated with the following rewrite.

$$
\text{[spill]} \qquad \mathbf{spill}[v,\ s] \longleftrightarrow \mathbf{spill}[s]
$$

4.6 Assembly optimization

There are several simple optimizations that can be performed on the generated assembly, including dead-code elimination and reserve coalescing. Dead-code elimination has a simple specification: any instruction that defines a new binding variable can be eliminated if the variable is never used. The following rewrites capture this property.

$$
\begin{array}{ll}
\text{[dmov]} & \mathit{MOV}\ o,\ \lambda v.e \longleftrightarrow e \\
\text{[dinst1]} & \mathit{inst1}\ o_r,\ \lambda v.e \longleftrightarrow e \\
\text{[dinst2]} & \mathit{inst2}\ o,\ o_r,\ \lambda v.e \longleftrightarrow e \\
\text{[dinst3]} & \mathit{inst3}\ o,\ o_{r_1},\ o_{r_2},\ \lambda v_1, v_2.e \longleftrightarrow e
\end{array}
$$

As we mentioned in Section 3.4, this kind of dead-code elimination should not be applied if the instruction being eliminated can raise an exception.

Another useful optimization is the coalescing of **reserve**(i) instructions, which call the garbage collector if i bytes of storage are not available. In the current version of the language, all reservations specify a constant number of bytes of storage, and these reservations can be propagated up the expression tree and coalesced. The first step is an upward propagation of the reserve statement. The following rewrites illustrate the process.

$$
\begin{array}{ll}
\text{[rmov]} & MOV\ o,\ \lambda v.\mathbf{reserve}(i); e[v] \\
\longleftrightarrow & \mathbf{reserve}(i); MOV\ o,\ \lambda v.e[v] \\
\text{[rinst2]} & \mathit{inst2}\ o,\ o_r,\ \lambda v.\mathbf{reserve}(i); e[v] \\
\longleftrightarrow & \mathbf{reserve}(i); \mathit{inst2}\ o,\ o_r,\ \lambda v.e[v]
\end{array}
$$

Adjacent reservations can also be coalesced.

$$
\text{[rres]} \quad \mathbf{reserve}(i_1); \mathbf{reserve}(i_2); e \longleftrightarrow \mathbf{reserve}(i_1 + i_2); e
$$

Two reservations at a conditional boundary can also be coalesced. To ensure that both branches have a reserve, it is always legal to introduce a reservation for 0 bytes of storage.

$$
\begin{array}{ll}
\text{[rif]} & Jcc\ \mathbf{then}\ \mathbf{reserve}(i_1); e_1\ \mathbf{else}\ \mathbf{reserve}(i_2); e_2 \\
\longleftrightarrow & \mathbf{reserve}(max(i_1; i_2)); Jcc\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 \\
\text{[rzero]} & e \longleftrightarrow \mathbf{reserve}(0); e
\end{array}
$$

5. SUMMARY AND FUTURE WORK

One of the points we have stressed in this presentation is that the implementation of formal compilers is easy, perhaps easier than traditional compiler development using a general-purpose language. This case study presents a convincing argument based on the authors' previous experience implementing compilers using traditional methods. The formal process was easier to specify and implement, and MetaPRL provided a great deal of automation for frequently occurring tasks. In most cases, the implementation of a new compiler phase meant only the development of new rewrite rules. There is very little of the "grunge" code that plagues traditional implementations, such as the maintenance of tables that keep track of the variables in scope, code-walking procedures to apply a transformation to the program's subterms, and other kinds of housekeeping code.

As a basis of comparison, we can compare the formal compiler in this paper to a similar native-code compiler for a fragment of the Java language we developed as part of the Mojave project [7]. The Java compiler is written in OCaml, and uses an intermediate representation similar to the one presented in this paper, with two main differences: the Java intermediate representation is typed, and the x86 assembly language is not scoped.

Figure 9 gives a comparison of some of the key parts of both compilers in terms of lines of code, where we omit code that implements the Java type system and class constructs. The formal compiler columns list the total lines of code for the term rewrites, as well as the total code including rewrite strategies. The size of the total code base in the formal compiler is still quite large due to the extensive code needed to implemented the graph coloring algorithm for the register allocator. Preliminary tests suggest that performance of programs generated from the formal compiler is comparable, sometimes better than, the Java compiler due to a better spilling strategy.

Description	Formal compiler		Java
	Rewrites	Total	
CPS conversion	44	347	338
Closure conversion	54	410	1076
Code generation	214	648	1012
Total code base	484	10000	12000

Figure 9: Code comparison

The work presented in this paper took roughly one person-week of effort from concept to implementation, while the Java implementation took roughly three times as long. It should be noted that, while the Java compiler has been stable for about a year, it still undergoes periodic debugging. Register allocation is especially problematic to debug in the Java compiler, since errors are not caught at compile time, but typically cause memory faults in the generated program.

This work is far from complete. The current example serves as a proof of concept, but it remains to be seen what issues will arise when the formal compilation methodology is applied to more complex programming languages. For future work, we intend to approach the problem of developing and validating formal compilers in three steps. The first step is the development of typed intermediate languages. These languages admit a broader class of rewrite transformations that are conditioned on well-typed programs, and the typed language serves as a launching point for compiler validation. The second step is to develop a semantics of the intermediate language and validate the rewrite rules for a small source language similar to the one presented here. It is not clear whether the same properties should be applied to the assembly language—whether the assembly language should be typed, and whether it is feasible to develop a simple formal model of the target architecture that will allow the code generation and register allocations phases to be verified. The final step is to extend the source language to one resembling a modern general-purpose language.

6. RELATED WORK

The use of higher-order abstract syntax, logical environments, and term rewriting for compiler implementation and validation are not new areas individually.

Term rewriting has been successfully used to describe programming language syntax and semantics, and there are systems that provide efficient term representations of programs as well as rewrite rules for expressing program transformations. For instance, the `ASF+SDF` environment [11] allows the programmer to construct the term representation of a wide variety of programming syntax and to specify equations as rewrite rules. These rewrites may be conditional or unconditional, and are applied until a normal form is reached. Using equations, programmers can specify optimizations, program transformations, and evaluation. The `ASF+SDF` system targets the generation of informal rewriting code that can be used in a compiler implementation.

Liang [10] implemented a compiler for a simple imperative language using a higher-order abstract syntax implementation in λProlog. Liang's approach includes several of the phases we describe here, including parsing, CPS conversion, and code generation using a instruction set defined using higher-abstract syntax (although in Liang's case, registers are referred to indirectly through a meta-level store, and we represent registers directly as variables). Liang does not address the issue of validation in this work, and the primary role of λProlog is to simplify the compiler implementation. In contrast to our approach, in Liang's work the entire compiler was implemented in λProlog, even the parts of the compiler where implementation in a more traditional language might have been more convenient (such as register allocation code).

FreshML [17] adds to the ML language support for straightforward encoding of variable bindings and alpha-equivalence classes. Our approach differs in several important ways. Substitution and testing for free occurrences of variables are explicit operations in FreshML, while MetaPRL provides a convenient implicit syntax for these operations. Binding names in FreshML are inaccessible, while only the formal parts of MetaPRL are prohibited from accessing the names. Informal portions—such as code to print debugging messages to the compiler writer, or warning and error messages to the compiler user—can access the binding names, which aids development and debugging. FreshML is primarily an effort to add automation; it does not address the issue of validation directly.

Previous work has also focused on augmenting compilers with formal tools. Instead of trying to split the compiler into a formal part and a heuristic part, one can attempt to treat the *whole* compiler as a heuristic adding some external code that would watch over what the compiler is doing and try to establish the equivalence of the intermediate and final results. For example, the work of Necula and Lee [13, 14] has led to effective mechanisms for certifying the output of compilers (e.g., with respect to type and memory-access safety), and for verifying that intermediate transformations on the code preserve its semantics. While these approaches certify the code and ease the debugging process (errors can be flagged at compile time rather than at run-time), it is not clear that they simplify the implementation task.

There have been efforts to present more functional accounts of assembly as well. Morrisett *et. al.* [12] developed a typed assembly language capable capable of supporting many high-level programming constructs and proof carrying code. In this scheme, well-typed assembly programs cannot "go wrong."

7. REFERENCES

[1] A. W. Appel. *Compiling with Continuations.* Cambridge University Press, 1992.

[2] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein. Register allocation via coloring. *Computer Languages*, 6(1):47–57, Jan. 1981.

[3] A. Granicz and J. Hickey. Phobos: A front-end approach to extensible compilers. In *36^{th} Hawaii International Conference on System Sciences.* IEEE, 2002.

[4] J. Hickey, A. Nogin, R. L. Constable, B. E. Aydemir, E. Barzilay, Y. Bryukhov, R. Eaton, A. Granicz, A. Kopylov, C. Kreitz, V. N. Krupski, L. Lorigo, S. Schmitt, C. Witty, and X. Yu. MetaPRL — a modular logical environment. Submitted to the TPHOLs 2003 Conference, 2003.

[5] J. Hickey, J. D. Smith, B. Aydemir, N. Gray, A. Granicz, and C. Tapus. Process migration and transactions using a novel intermediate language. Technical Report caltechCSTR:2002.007, California Institute of Technology, Computer Science, August 2002.

[6] J. J. Hickey. *The MetaPRL Logical Programming Environment.* PhD thesis, Cornell University, Ithaca, NY, Jan. 2001.

[7] J. J. Hickey et al. Mojave research project home page. http://mojave.caltech.edu/.

[8] J. J. Hickey, A. Nogin, A. Kopylov, et al. MetaPRL home page. http://metaprl.org/.

[9] S. C. Johnson. Yacc — yet another compiler compiler. Computer Science Technical Report 32, AT&T Bell Laboratories, July 1975.

[10] C. C. Liang. Compiler construction in higher order logic programming. In *Practical Aspects of Declarative Languages*, volume 2257 of *Lecture Notes in Computer Science*, pages 47–63, 2002.

[11] M. G. J. van den Brand, J. Heering, P. Klint, and P. A. Olivier. Compiling Rewrite Systems: The ASF+SDF Compiler. *ACM Transactions on Programming Languages and Systems*, 24:334–368, 2002.

[12] J. G. Morrisett, D. Walker, K. Crary, and N. Glew. From system F to typed assembly language. *Principles of Programming Languages*, 1998.

[13] G. C. Necula. Translation validation for an optimizing compiler. *ACM SIGPLAN Notices*, 35(5):83–94, 2000.

[14] G. C. Necula and P. Lee. The design and implementation of a certifying compiler. In *Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 333–344, 1998.

[15] A. Nogin and J. Hickey. Sequent schema for derived rules. In V. A. Carreño, C. A. Muñoz, and S. Tahar, editors, *Proceedings of the 15^{th} International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2002)*, volume 2410 of *Lecture Notes in Computer Science*, pages 281–297. Springer-Verlag, 2002.

[16] F. Pfenning and C. Elliott. Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation (PLDI)*, volume 23(7) of *SIGPLAN Notices*, pages 199–208, Atlanta, Georgia, June 1988. ACM Press.

[17] A. M. Pitts and M. Gabbay. A metalanguage for programming with bound names modulo renaming. In R. Backhouse and J. N. Oliveira, editors, *Mathematics of Program Construction*, volume 1837 of *Lecture Notes in Computer Science*, pages 230–255. Springer-Verlag, Heidelberg, 2000.

[18] D. Tarditi. *Design and implementation of code optimizations for a type-directed compiler for Standard ML.* PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1997.

[19] J. D. Ullman. *Elements of ML Programming.* Prentice Hall, 1998.

[20] P. Weis and X. Leroy. *Le langage Caml.* Dunod, Paris, 2nd edition, 1999. In French.