© 2005 *Kluwer Academic Publishers. Printed in the Netherlands.*

Formal Compiler Construction in a Logical Framework

Jason Hickey* and Aleksey Nogin*
California Institute of Technology
1200 E. California Blvd.
Pasadena, CA 91125, USA
{jyh,nogin}@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.

Keywords: Formal compiler, higher-order abstract syntax, logical programming environment

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 [Ull98],

* 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.

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 [NH02, PE88] and term rewriting in an existing 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 higher-order abstract syntax and 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 [NH02]. 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 compilation 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, have to be split into a sequence of "small step" term rewrites, which be difficult at times.

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 [HNC+03, Hic01, HNK+]. The compiler is fully implemented and online as part of the Mojave research project [H+]. 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/`.

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 [Ull98]. 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. To give a better idea of what it takes to implement a compiler using our approach, we also provide an Appendix with a pretty-printed annotated version of some of the relevant source code.

The high-level outline of the paper is as follows.

- Section 2 Language
- Section 3 Intermediate representation (IR)
- Section 4 Intel x86 assembly code generation
- Section 5 Summary and future work
- Section 6 Related work
- Appendix Example source code

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 [NH02]. 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.

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

The left-hand-side of the rewrite is a pattern called the *redex*. The $v_1[x]$ stands for an arbitrary term that may have free occurrences of the 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.

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

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.

Rewrites provide a mechanism for transforming programs, but they are not self-directed. LCF-style [GMW79] *tactics* direct the compilation process, deciding when and where to apply rewrites. MetaPRL's tactic language is OCaml [WL99]. 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 rewrites is then written in OCaml, using a rich set of primitives provided by MetaPRL.

Tactic code can use any possible technique for deciding which rule or rewrite to apply next, including inspecting the program intentionally, manufacturing terms, or even consulting oracles. However, the *only* way for a tactic to manipulate the compilation is by applying rules or rewrites. A beneficial consequence of this is that if all the rules and rewrites in the compiler are semantics-preserving, then tactics need not be trusted. Under these circumstances, errors in tactics can prevent compilation progress, but they cannot corrupt a compilation. The desire to keep rules and rewrites semantics-preserving is one of the primary design goals in our methodology.

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 $\Gamma \vdash \textbf{compilable}(e)$, which states the the program e is compilable in any logical context Γ. The meaning of the $\textbf{compilable}(e)$ judgment is defined by the target architecture. A program e' is compilable if it is equivalent to 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. Language

The abstract syntax of the language of our compiler is shown in Figure 1. In order to use the formal system for program transformation, the concrete syntax of the source-level programs must first be translated into a term representation for use in the MetaPRL framework. We achieve that by using the Phobos [GH02] extensible lexer and parser, which is a part of the framework. A Phobos language specification resembles a typical parser definition in YACC [Joh75], except that semantic actions for productions use the MetaPRL term rewriting engine.

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 representations we use throughout this work are variants of A-normal form [FSDF93]. These representations are similar to the functional intermediate representations used by several groups [App92, HSA$^+$02, Tar97], 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.

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

Figure 1. Program syntax

Function definitions deserve special mention. Functions are defined using the **let rec** $R = d$ **in** e term, where d is a list of mutually recursive functions, and variable 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 use, principally because it violates the rule of higher-order abstract syntax: namely, that (function) variables be represented as variables in the meta-language. We are currently investigating the use of *sequents* to represent mutual recursion in order to address these problems.

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. We introduce an administrative term $\texttt{IR}\{e_1; v.e_2[v]\}$ (displayed as $[\![e_1]\!]_{IR} v.e_2[v]$) that

$binop ::= + \mid - \mid * \mid /$	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
$e_\lambda ::= \lambda v.e_\lambda \mid \lambda v.e$	Functions
$d ::= \mathbf{fun}\ l = e_\lambda\ \mathbf{and}\ d$	Function definitions
$\mid\ \epsilon$	

Figure 2. Intermediate Representation

represents the translation of an expression e_1 to an IR atom. The second argument ($e_2[v]$) is a *meta-continuation* of the translation process. In other words, e_2 represents *the rest* of the program and v marks the location where the IR for e_1 would go.

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. Note that all the rules perform substitution, which is specified implicitly using higher-

order abstract syntax.

$$
\begin{array}{ll}
\text{[int]} & [\![i]\!]_{IR} v.e[v] \longleftrightarrow e[i] \\[1em]
\text{[var]} & [\![v_1]\!]_{IR} v_2.e[v_2] \longleftrightarrow e[v_1] \\[1em]
\text{[add]} & [\![e_1 + e_2]\!]_{IR} v.e[v] \\
\longleftrightarrow & [\![e_1]\!]_{IR} v_1.[\![e_2]\!]_{IR} v_2.e[v_1 + v_2] \\[1em]
\text{[set]} & [\![e_1.[e_2] \leftarrow e_3]\!]_{IR} v.e_4[v] \\
\longleftrightarrow & [\![e_1]\!]_{IR} v_1. \\
 & [\![e_2]\!]_{IR} v_2. \\
 & [\![e_3]\!]_{IR} v_3. \\
 & v_1.[v_2] \leftarrow v_3; \\
 & e_4[\bot]
\end{array}
$$

Here [int] and [var] specify that variables and numerical constants do not have to be further translated — so we simply pass the original variable or numerical constant to meta-continuation. The [add] rewrite specifies that in order to translate $e_1 + e_2$, we need to first translate e_1 (passing the result as v_1), continuing with translation of e_2 (passing the result as v_2), continuing with passing the IR expression $v_1 + v_2$ to the original meta-continuation.

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.

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

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.

$$
\begin{array}{ll}
\text{[letrec]} & [\![\, \mathbf{let\ \ rec}\ R = d\ \mathbf{in}\ e_1]\!]_{IR} v.e_2[v] \\
\longleftrightarrow & \mathbf{let\ \ rec}\ R = [\![d]\!]_{IR}\ \mathbf{in}\ [\![e_1]\!]_{IR} v.e_2[v] \\[1em]
\text{[fun]} & [\![\mathbf{fun}\ l = e\ \mathbf{and}\ d]\!]_{IR} \\
\longleftrightarrow & \mathbf{fun}\ l = [\![e]\!]_{IR} v.\mathbf{return}\ v\ \mathbf{and}\ [\![d]\!]_{IR}
\end{array}
$$

$$
\begin{array}{ll}
\text{[param]} & [\![\lambda_p v_1.e_1[v_1]]\!]_{IR} v_2.e_2[v_2] \\
\longleftrightarrow & \lambda v_1.([\![e_1[v_1]]\!]_{IR} v_2.e_2[v_2])
\end{array}
$$

$$
\begin{array}{ll}
\text{[abs]} & [\![\lambda v_1.e_1[v_1]]\!]_{IR} v_2.e_2[v_2] \\
\longleftrightarrow & \textbf{let rec } R = \\
& \quad \textbf{fun } g = \lambda v_1.[\![e_1[v_1]]\!]_{IR} v_3.\textbf{return } v_3 \textbf{ and } \epsilon \\
& \textbf{in } e_2[R.g]
\end{array}
$$

All the rewrites for the AST to IR translation are automatically collected by the MetaPRL system into a syntax-directed lookup table (each rewrite is annotated with the name of the appropriate table) and creates the tactic for sweeping the program and performing all the applicable transformations [HN04].

3.2. CPS conversion

CPS conversion is a 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 **let** $v = a_1(a_2)$ **in** e and **return** a are eliminated. The main objective in CPS conversion is to pass the result of the computation to a continuation function.

CPS conversion is not a requsite part of our methodology. However, it represents an important style of transformation, and therefore we choose to illustrate it in this case study.

There are different ways of formalizing the CPS conversion (see Section 5 for a discussion). In this compiler we used the following inference rule, which states that a program e is compilable if for all functions c, the program $[\![e]\!]_c$ is compilable.

$$
\text{[cps_prog]} \quad \frac{\Gamma, c\colon \mathit{exp} \vdash \textbf{compilable}([\![e]\!]_c)}{\Gamma \vdash \textbf{compilable}(e)}
$$

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 in HOAS setting 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 $\mathbf{let\,rec}\, R \;\mathbf{with}\; [Fr = ()] \;\; = d \;\mathbf{in}\; e$, where Fr 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 $R(Fr)$, which represents the partial application of the record R to the tuple Fr.

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

The new $\mathbf{let\,rec}\, R \;\mathbf{with}\; [Fr = f] \;\; = d \;\mathbf{in}\; e[Fr]$ expression is an administrative term that helps us keep track of the progress of the

closure conversion; it will be eliminated from the program by the end of the closure conversion.

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

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

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

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

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

$$\begin{array}{ll}
\text{[arg]} & \mathbf{frame}(\mathit{Fr}, \mathbf{fun}\ l = \lambda v.e[\mathit{Fr}; v]\ \mathbf{and}\ d[\mathit{Fr}]) \\
\longleftrightarrow & \mathbf{fun}\ l = \lambda \mathit{Fr}.\lambda v.e[\mathit{Fr}; v]\ \mathbf{and\ frame}(\mathit{Fr}, d[\mathit{Fr}])
\end{array}$$

$$\begin{array}{ll}
\text{[sub]} & \mathbf{let}\, v_1 = a_1.[a_2]\ \mathbf{in} \\
& \mathbf{fun}\ l = \lambda v_2.e[v_1; v_2]\ \mathbf{and}\ d[v_1] \\
\longleftrightarrow & \mathbf{fun}\ l = \lambda v_2.\,\mathbf{let}\, v_1 = a_1.[a_2]\ \mathbf{in}\ e[v_1; v_2]\ \mathbf{and} \\
& \mathbf{let}\, v_1 = a_1.[a_2]\ \mathbf{in}\ d[v_1]
\end{array}$$

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 4.3, 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 *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 *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

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

Figure 3. Scoped Intel x86 instruction set

variable being modified. Instructions that *do not* modify registers use the traditional non-binding form of the instruction. For example, the instruction $ADD\ v_1\textbf{,}\ (\%v_2); e$ 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 on the next page (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.
- 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 $MOV\ o_1,\ \lambda v_2.e[v_2]$ copies the value in operand o_1 to variable v_2.
- One-operand instructions have the forms $inst1\ o_1; e$ (where o_1 must be an indirect operand), and $inst1\ v_1,\ \lambda v_2.e$. For example, the instruction $INC\ (\%r_1); e$ performs the operation $(\%r_1) \leftarrow (\%r_1) + 1; e$; and the instruction $INC\ \%r_1,\ \lambda r_2.e$ performs the operation $\textbf{let}\, r_2 = r_1 + 1\ \textbf{in}\ e$.
- Two-operand instructions have the forms $inst2\ o_1,\ o_2; e$, where o_2 must be an indirect operand; and $inst2\ o_1,\ v_2,\ \lambda v_3.e$. For example, the instruction $ADD\ \%r_1,\ (\%r_2); e$ performs the operation $(\%r_2) \leftarrow (\%r_2) + r_1; e$; and the instruction $ADD\ o_1,\ v_2,\ \lambda v_3.e$ 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 $inst3\ o_1,\ v_2,\ v_3,\ \lambda v_4, v_5.e$. For example, the instruction $DIV\ \%r_1,\ \%r_2,\ \%r_3,\ \lambda r_4, r_5.e$ 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 (eax and edx are two specific processor registers).

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

- The comparison operand has the form $CMP\ o_1,\ o_2; e$, 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 greater flexibility during code-motion optimizations on the assembly.
- The unconditional branch operation $JMP\ o(o_1, \ldots, o_n)$ 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 calling convention is described in the next section).
- The conditional branch operation Jcc **then** e_1 **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 **let rec** $R = d$ **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. The runtime environment

Before generating code, we must consider the role of the runtime. There are two important parts to consider, including data representation and memory management, and the calling convention for functions.

4.1.1. *Heap representation and garbage collection*

Since the source language contains first-class functions (and we have introduced continuations as well), the most straightforward approach to memory management is to use a garbage collector. We adopt a data representation similar to that used in the Objective Caml runtime [Ler97], where all heap data has one of two forms, it is either 1) a block of memory, with a header word that specifies the size of the block, or 2) it is a single machine word that specifies an integer. Furthermore, we adopt the OCaml convention that all blocks are aligned to machine-word boundaries, and integer values have 31 significant bits, where the least significant bit in the machine word is always 1. A diagram of these values is shown in Figure 4.

These conventions provide run-time tags for the garbage collector. Given a machine word that represents a heap-allocated value, the value is an integer if the least-significant-bit is set, otherwise it is a pointer to a heap-allocated block.

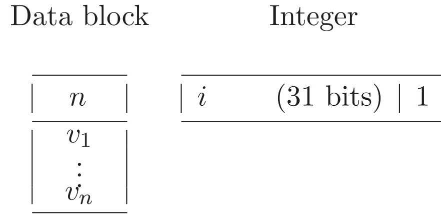

Figure 4. Runtime data representation

In this paper, we assume that the garbage collector is trusted.

4.1.2. *Calling convention*

The second runtime issue of interest is the calling convention. As specified by our instruction set, functions e_λ are specified together with their parameters, and branches *JMP* $o(o_1, \ldots, o_n)$ specify the arguments to the function call. The purpose of the calling convention is to ensure that the locations of the arguments in the call are the same as the locations of the parameters to the function.

The particular locations do not matter, they just need to be the same. The register allocator (Section 4.4) is given the task of ensuring that the calling convention is followed, and that the arguments are passed in the expected locations. For calls to external functions (for example, for input/output), we adopt the policy of passing all arguments on the stack.

4.2. Translation to concrete assembly

Perhaps the first question to consider is how to generate concrete machine code from the HOAS representation. As mentioned previously, 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.

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 *inst1* $o_m; e$, *inst2* o_r, $o_m; e$, and *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

$$\mathit{inst1}\ o_r\textbf{,}\ \lambda r.e,$$

if $o_r = \%r$, then the instruction is implemented as *inst1* 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 additional `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.

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

– 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.

4.3. 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.

$$
\begin{array}{ll}
\text{[false]} & [\![\bot]\!]_{\mathbf{a}} v.e[v] \longleftrightarrow e[\$1] \\
\text{[true]} & [\![\top]\!]_{\mathbf{a}} v.e[v] \longleftrightarrow e[\$3] \\
\text{[int]} & [\![i]\!]_{\mathbf{a}} v.e[v] \longleftrightarrow e[\$i * 2 + 1] \\
\text{[var]} & [\![v_1]\!]_{\mathbf{a}} v_2.e[v_2] \longleftrightarrow e[\%v_1] \\
\text{[label]} & [\![R.l]\!]_{\mathbf{a}} v.e[v] \longleftrightarrow e[\$R.l]
\end{array}
$$

$$
\begin{array}{ll}
\text{[add]} & [\![a_1 + a_2]\!]_{\mathbf{a}} v.e[v] \\
\longleftrightarrow & [\![a_1]\!]_{\mathbf{a}} v_1. \\
 & [\![a_2]\!]_{\mathbf{a}} v_2. \\
 & \mathit{ADD}\ v_2,\ v_1,\ \lambda tmp. \\
 & \mathit{DEC}\ \%tmp,\ \lambda sum. \\
 & e[\%sum]
\end{array}
$$

$$
\begin{array}{ll}
\text{[div]} & [\![a_1\ /\ a_2]\!]_{\mathbf{a}} v.e[v] \\
\longleftrightarrow & [\![a_1]\!]_{\mathbf{a}} v_1. \\
 & [\![a_2]\!]_{\mathbf{a}} v_2. \\
 & \mathit{SAR}\ \$1,\ v_1,\ \lambda v_1'. \\
 & \mathit{SAR}\ \$1,\ v_2,\ \lambda v_2'. \\
 & \mathit{MOV}\ \$0,\ \lambda v_3. \\
 & \mathit{DIV}\ \%v_1',\ \%v_2',\ \%v_3',\ \lambda q', r'. \\
 & \mathit{SHL}\ \$1,\ \%q',\ \lambda q''. \\
 & \mathit{OR}\ \$1,\ \%q'',\ \lambda q. \\
 & e[\%q]
\end{array}
$$

Figure 5. Translation of atoms to x86 assembly

We express these translations with the term $[\![e]\!]_{\mathbf{a}}$, which is the translation of the IR expression e to an assembly expression; and $[\![a]\!]_{\mathbf{a}} v.e[v]$, which produces the assembly operand for the atom a and substitutes it for the variable v in assembly expression $e[v]$.

4.3.1. *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 5. As mentioned in Section 4.1.1, we use a 31-bit representation of integers, where the least-significant-bit is always set to 1. 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.

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

Figure 6. Translation of expressions to x86 assembly

4.3.2. *Expression translation*

Expressions translate to sequences of assembly instructions. A representative set of translations in shown in Figure 6. The translation of $\mathbf{let}\, v = a$ **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.

The memory operations shown in Figure 7 are among the most complicated translations. By convention, 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

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

Figure 7. Translation of memory operations to x86 assembly

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.

The `sub` rule shows the translation of an array subscripting operation. Here 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**). There is a similar rule for projecting values from the tuples for closure environments, where the bounds-check may be omitted.

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 8 shows the implementation of some of the helper terms: the **reserve**(i) expression determines

$$
\begin{array}{ll}
\text{[reserve]} & \mathbf{reserve}(i); e \\
\longleftrightarrow & \mathit{MOV}\ \mathbf{context}[\mathit{limit}]\mathbf{,}\ \lambda \mathit{limit}. \\
& \mathit{SUB}\ \mathbf{context}[\mathit{next}]\mathbf{,}\ \%\mathit{limit}\mathbf{,}\ \lambda \mathit{free}. \\
& \mathit{CMP}\ i\mathbf{,}\ \%\mathit{free} \\
& J\mathbf{b}\ \mathbf{then}\ \mathit{gc}(i)\ \mathbf{else}\ e
\end{array}
$$

$$
\begin{array}{ll}
\text{[stuple1]} & \mathbf{store_tuple}(p, i, (a :: \mathit{args})); e \\
\longleftrightarrow & [\![a]\!]_{\mathbf{a}} v. \\
& \mathit{MOV}\ v\mathbf{,}\ i(\%p) \\
& \mathbf{store_tuple}(p, i+4, \mathit{args}); e
\end{array}
$$

$$
\begin{array}{ll}
\text{[stuple2]} & \mathbf{store_tuple}(p, i, ()); e \longleftrightarrow e
\end{array}
$$

$$
\begin{array}{ll}
\text{[copy1]} & \mathbf{copy_args}((a :: \mathit{args}), \mathit{vargs})\lambda v.e[v] \\
\longleftrightarrow & [\![a]\!]_{\mathbf{a}} v'. \\
& \mathit{MOV}\ v'\mathbf{,}\ \lambda v. \\
& \mathbf{copy_args}(\mathit{args}, (\%v :: \mathit{vargs}))\lambda v.e[v]
\end{array}
$$

$$
\begin{array}{ll}
\text{[copy2]} & \mathbf{copy_args}((), \mathit{vargs})\lambda v.e[v] \\
\longleftrightarrow & e[\mathbf{reverse}(\mathit{vargs})]
\end{array}
$$

Figure 8. Auxiliary terms for x86 code generation

whether sufficient storage is present for an allocation of i bytes, and calls the garbage collector otherwise; the $\mathbf{store_tuple}(p, i, \mathit{args}); e$ term generates the code to initialize the fields of a tuple from a set of arguments; and the $\mathbf{copy_args}(\mathit{args}, \mathit{vargs})\lambda v.e$ term copies the argument list in *args* into registers.

4.4. 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. Because we are using higher-order abstract syntax, 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 problem, and the majority of the code in our implementation is devoted to a Chaitin-style [CAC+81] 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.

4.5. 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.

$$[\text{smov}] \quad MOV\ o,\ \lambda v.e[v] \longleftrightarrow MOV\ o,\ \lambda spill_i.e[spill_i]$$

$$\begin{array}{ll} [\text{sinst2}] & inst2\ o,\ o_r,\ \lambda v.e[v] \\ \longleftrightarrow & MOV\ o_r,\ \lambda spill_i. \\ & inst2\ o,\ spill_i \\ & e[spill_i] \end{array}$$

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 9. The instruction $SPILL\ o_r,\ \lambda s.e[s]$ generates a new spill location represented in the variable s, and stores the operand o_r in that spill location. The operand $\mathbf{spill}[v,\ s]$ 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

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

Figure 9. Spill pseudo-operands and instructions

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

Figure 10. Spill example

same. The operand $\mathbf{spill}[s]$ 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 $\mathbf{spill}[v,\ s]$. For example, consider the code fragment shown in Figure 10, and suppose the register allocator determines that the variable v is to be spilled, because a register cannot be assigned in code segment 2.

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, as shown in Figure 10. Note that the later uses refer to the new registers, creating a copying daisy-chain, but the registers have not been actually eliminated.

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 $\mathbf{spill}[v_2,\ s]$ operand is replaced with $\mathbf{spill}[s]$ 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.

4.6. 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 $\mathbf{spill}[v,\ s]$ with $\mathbf{spill}[s]$ when requested by the register allocator.

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 $\mathbf{spill}[v,\ s]$, 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. $i(\%v)$.

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

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

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 $\mathbf{spill}[v_1,\ s]$, and translates it to a new program that fetches the spill into a new register v_2 and uses the new spill operand $\mathbf{spill}[v_2,\ s]$ in the remainder of the program. This rewrite is selectively applied before any instruction that uses an operand $\mathbf{spill}[v_1,\ s]$.

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

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

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

4.7. 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.

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

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 $\textbf{reserve}(i)$ 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.

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

Adjacent reservations can also be coalesced.

$$
\begin{array}{ll}
\text{[rres]} & \textbf{reserve}(i_1); \textbf{reserve}(i_2); e \\
\longleftrightarrow & \textbf{reserve}(i_1 + i_2); e
\end{array}
$$

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.

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

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 [H+]. 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 11 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.

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. We are currently working on the construction of

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 11. Code comparison

a compiler for a *typed* language, using sequent notation to address the problem of retaining higher order abstract syntax in the definition of mutually recursive functions.

In the compiler presented in this paper we took a very conservative approach to making sure the the rule rewrite transformations do not affect the program semantics. A very good example of this is the CPS transformation (Section 3.2). There we have defined the semantics of the $[\![e]\!]_c$ to just be the function c applied to expression e. Under this semantics, the rule `cps_prog` (that states that a program is compilable if the result of its CPS transformation is compilable) is obviously valid — if we take c to be an identity function, then under this semantics the rule simply does not change the program. This semantics also provides us with sufficient information to be able to separately validate each individual CPS-related program transformation.

A downside of such a conservative approach is that it becomes very hard to write transformations in an optimal way. In particular, the CPS rewrites presented is this paper introduce a large number of "administrative" beta-redices that would need to be eliminated in subsequent optimization stages. We can chose an alternative approach where the CPS term is defined as performing a syntactical transformation of a program. In this approach, all the rewrites for the CPS term become simply parts of the *definition* of the CPS transformation. All the work required to prove that the transformation does not change the meaning of the program goes into establishing that the corresponding `cps_prog` rule is valid. This approach makes it much easier to specify the CPS transformation in an optimal way, following the approach of Danvy and Fellinski [DF92]. We currently use this approach in our work-in-progress compiler [GHNŢ04, HNG05]; the specification of the CPS transformation ends up being even simpler than Danvy and Fellinski's because of the efficiency of the HOAS language that we use.

This paper can be considered to be a first step in a much larger project. One of our main goals for this step was to investigate the feasibility of this approach in a small case study. We believe that we have demonstrated that at least on this level the approach we propose is definitely feasible. In fact, almost every time the reality of this work did not match our expectations it was because this approach turned out to be *easier* than we have originally anticipated. Now that this case study have demonstrated the feasibility of this approach *in principle*, we have moved on to implementing a more realistic compiler for a strongly-typed ML-like language [GHNŢ04, HNG05].

This second-generation formal compiler is already implemented, for the most part. In addition to taking advantage of the lessons learned in this case study (such as using Danvy an Fellinski's approach to CPS and using nested sequents [GHNŢ04] to represent recursive functions), one of the main goals of the second-generation compiler work was to explore the issues of *modularity* and *feature isolation*. This goal was successfully achieved as well — we were able to structure the compiler in such a way that different language features are cleanly isolated and experimentation with one of the language features can not break compilation of unrelated features.

The fact that in our approach all the program transformations are *individually* semantics preserving, together with feature isolation and modularity of the second-generation compiler makes our compilers readily amenable to incremental verification (including both on-paper verification and computer-aided formal proofs). While verification was not among the goals of our compiler case studies, it is among the goals of the larger project. It is also one of our larger goals to explore formally the issues of correctness and programming language meta-theory. In a related effort [NKYH05], we are investigating the use of reflection as a means for meta-reasoning about formal artifacts. We expect that reflection will provide a generic mechanism for automatically internalizing the artifacts specified in a prover, including those presented here, and programming language meta-theory in general.

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 [vdBHKO02] 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.

`FreshML` [PG00] 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.

Liang [Lia02] 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).

Hannan and Pfenning [HP92] constructed a verified compiler in LF (as realized in the Elf programming language) for the untyped lambda calculus and a variant of the CAM [CCM87] runtime. This work formalizes both compiler transformation and verifications as deductive systems, and verification is against an operational semantics.

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 [Nec00, NL98]

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. Pnueli, Siegel, and Singerman [PSS98] perform verification in a similar way, not by validating the compiler, but by validating the result of a transformation using simulation-based reasoning.

Semantics-directed compilation [Lee89] is aimed at allowing language designers to generate compilers from high-level semantic specifications. Although it has some overlap with our work, it does not address the issue of trust in the compiler. No proof is generated to accompany the compiler, and the compiler generator must be trusted if the generated compiler is to be trusted.

Boyle, Resler, and Winter [BRW97], outline an approach to building trusted compilers that is similar to our own. Like us, they propose using rewrites to transform code during compilation. Winter develops this further in the HATS system [Win99] with a special-purpose transformation grammar. An advantage of this approach is that the transformation language can be tailored for the compilation process. However, this significantly restricts the generality of the approach, and limits re-use of existing methods and tools.

There have been efforts to present more functional accounts of assembly as well. Morrisett *et. al.* [MWCG98] developed a typed assembly language capable of supporting many high-level programming constructs and proof carrying code. In this scheme, well-typed assembly programs cannot "go wrong."

References

App92. Andrew W. Appel. *Compiling with Continuations.* Cambridge University Press, 1992.

BRW97. J. Boyle, R. Resler, and K. Winter. Do you trust your compiler? Applying formal methods to constructing high-assurance compilers. In *High-Assurance Systems Engineering Workshop*, Washington, DC, August 1997.

CAC+81. Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. Register allocation via coloring. *Computer Languages*, 6(1):47–57, January 1981.

CCM87. G. Cousineau, P.L. Curien, and M. Mauny. The categorical abstract machine. *The Science of Programming*, 8(2):173–202, 1987.

DF92. Olivier Danvy and Andrzej Filinski. Representing control: A study of the CPS transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, 1992.

FSDF93. Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *Proceedings*

ACM SIGPLAN 1993 Conf. on Programming Language Design and Implementation, PLDI'93, Albuquerque, NM, USA, 23–25 June 1993, volume 28(6), pages 237–247. ACM Press, New York, 1993.

GH02. Adam Granicz and Jason Hickey. Phobos: A front-end approach to extensible compilers. In *36th Hawaii International Conference on System Sciences*. IEEE, 2002.

GHNŢ04. Nathaniel Gray, Jason Hickey, Aleksey Nogin, and Cristian Ţăpuş. Building extensible compilers in a formal framework. A formal framework user's perspective. In Konrad Slind, editor, *Emerging Trends. Proceedings of the 17th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2004)*, pages 57–70. University of Utah, 2004.

GMW79. Michael Gordon, Robin Milner, and Christopher Wadsworth. *Edinburgh LCF: a mechanized logic of computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, NY, 1979.

H+. Jason J. Hickey et al. Mojave research project home page. `http://mojave.caltech.edu/`.

Hic01. Jason J. Hickey. *The MetaPRL Logical Programming Environment*. PhD thesis, Cornell University, Ithaca, NY, January 2001.

HN04. Jason Hickey and Aleksey Nogin. Extensible hierarchical tactic construction in a logical framework. In Konrad Slind, Annette Bunker, and Ganesh Gopalakrishnan, editors, *Proceedings of the 17th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2004)*, volume 3223 of *Lecture Notes in Computer Science*, pages 136–151. Springer-Verlag, 2004.

HNC+03. Jason Hickey, Aleksey Nogin, Robert L. Constable, Brian E. Aydemir, Eli Barzilay, Yegor Bryukhov, Richard Eaton, Adam Granicz, Alexei Kopylov, Christoph Kreitz, Vladimir N. Krupski, Lori Lorigo, Stephan Schmitt, Carl Witty, and Xin Yu. MetaPRL — A modular logical environment. In David Basin and Burkhart Wolff, editors, *Proceedings of the 16th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2003)*, volume 2758 of *Lecture Notes in Computer Science*, pages 287–303. Springer-Verlag, 2003.

HNG05. Jason Hickey, Aleksey Nogin, and Nathaniel Gray. Programming language experimentation using proof assistants. Compiler development as a case study. To be submitted to Journal of Functional Programming (in preparation), 2005.

HNK+. Jason J. Hickey, Aleksey Nogin, Alexei Kopylov, et al. MetaPRL home page. `http://metaprl.org/`.

HP92. John Hannan and Frank Pfenning. Compiler verification in LF. In *Proceedings of the 7th Symposium on Logic in Computer Science*. IEEE Computer Society Press, 1992.

HSA+02. Jason Hickey, Justin D. Smith, Brian Aydemir, Nathaniel Gray, Adam Granicz, and Cristian Ţăpuş. Process migration and transactions using a novel intermediate language. Technical Report caltechCSTR:2002.007, California Institute of Technology, Computer Science, August 2002.

Joh75. Steven C. Johnson. Yacc — yet another compiler compiler. Computer Science Technical Report 32, AT&T Bell Laboratories, July 1975.

Lee89. Peter Lee. *Realistic compiler generation*. MIT Press, 1989.

Ler97. Xavier Leroy. *The Objective Caml system release 1.07.* INRIA, France, May 1997.

Lia02. Chuck 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.

MWCG98. J. Gregory Morrisett, David Walker, Karl Crary, and Neal Glew. From system F to typed assembly language. *Principles of Programming Languages*, 1998.

Nec00. George C. Necula. Translation validation for an optimizing compiler. *ACM SIGPLAN Notices*, 35(5):83–94, 2000.

NH02. Aleksey Nogin and Jason Hickey. Sequent schema for derived rules. In Victor A. Carreño, Cézar A. Muñoz, and Sophiène Tahar, editors, *Proceedings of the 15th 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.

NKYH05. Aleksey Nogin, Alexei Kopylov, Xin Yu, and Jason Hickey. A computational approach to reflective meta-reasoning about languages with bindings. In *MERLIN '05: Proceedings of the 3rd ACM SIGPLAN workshop on Mechanized reasoning about languages with variable binding*, pages 2–12. ACM Press, 2005. An extended version is available as California Institute of Technology technical report CaltechCSTR:2005.003.

NL98. George C. Necula and Peter 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.

PE88. Frank Pfenning and Conal 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.

PG00. Andrew M. Pitts and Murdoch 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.

PSS98. A. Pnueli, M. Siegel, and E. Singerman. Translation validation. *Lecture Notes in Computer Science*, 1384:151–166, 1998.

Tar97. David Tarditi. *Design and implementation of code optimizations for a type-directed compiler for Standard ML.* PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1997.

Ull98. Jeffrey D. Ullman. *Elements of ML Programming.* Prentice Hall, 1998.

vdBHKO02. Mark van den Brand, Jan Heering, Paul Klint, and Pieter A. Olivier. Compiling language definitions: The ASF+SDF compiler. *ACM Transactions of Programming Language Systems*, 24(4):334–368, July 2002.

Win99. Victor L. Winter. Program transformation in HATS. In *Proceedings of the Software Transformation Systems Workshop*, May 1999.

WL99. Pierre Weis and Xavier Leroy. *Le langage Caml.* Dunod, Paris, 2nd edition, 1999. In French.

Appendix

The following sections provide an example of the source code for the case study. The documentation is automatically generated by the Meta-PRL system from the source code.

A. M_ir module

This module defines the intermediate language for the M language. Here is the abstract syntax:

```
(* Values *)
v ::= i             (integers)
   |  b             (booleans)
   |  v             (variables)
   |  fun v -> e    (functions)
   |  (v1, v2)      (pairs)

(* Atoms (functional expressions) *)
a ::= i             (integers)
   |  b             (booleans)
   |  v             (variables)
   |  a1 op a2      (binary operation)
   |  fun x -> e    (unnamed functions)

(* Expressions *)
e ::= let v = a in e            (LetAtom)
   |  f(a)                      (TailCall)
   |  if a then e1 else e2      (Conditional)
   |  let v = a1.[a2] in e      (Subscripting)
   |  a1.[a2] <- a3; e          (Assignment)

      (* These are eliminated during CPS *)
   |  let v = f(a) in e         (Function application)
   |  return a
```

A program is a set of function definitions and an program expressed in a sequent. Each function must be declared, and defined separately.

A.1. Parents

Modules in MetaPRL are organized in a theory hierarchy. Each theory module extends its parent theories. In this case, the `M_ir` module extends base theories that define generic proof automation.

Extends `Base_theory`

A.2. Terms

The IR defines several binary operators for arithmetic and Boolean operations.

declare `M_ir!AddOp` **(displayed as** "*AddOp*"**)**
declare `M_ir!SubOp` **(displayed as** "*SubOp*"**)**
declare `M_ir!MulOp` **(displayed as** "*MulOp*"**)**
declare `M_ir!DivOp` **(displayed as** "*DivOp*"**)**
declare `M_ir!LtOp` **(displayed as** "*LtOp*"**)**
declare `M_ir!LeOp` **(displayed as** "*LeOp*"**)**
declare `M_ir!EqOp` **(displayed as** "*EqOp*"**)**
declare `M_ir!NeqOp` **(displayed as** "*NeqOp*"**)**
declare `M_ir!GeOp` **(displayed as** "*GeOp*"**)**
declare `M_ir!GtOp` **(displayed as** "*GtOp*"**)**

A.2.1. *Atoms*

Atoms represent expressions that are values: integers, variables, binary operations on atoms, and functions.

`AtomFun` is a lambda-abstraction, and `AtomFunVar` is the projection of a function from a recursive function definition (defined below).

declare `M_ir!AtomFalse` **(displayed as** "$false$"**)**
declare `M_ir!AtomTrue` **(displayed as** "$true$"**)**
declare `M_ir!AtomInt[i:n]` **(displayed as** "$\#i$"**)**
declare `M_ir!AtomBinop{'op; 'a1; 'a2}`
(displayed as "a_1 *op* a_2"**)**
declare `M_ir!AtomRelop{'op; 'a1; 'a2}`
(displayed as "a_1 *op* a_2"**)**
declare `M_ir!AtomFun{x. 'e['x]}` **(displayed as** "$\lambda_a x.\ e[x]$"**)**
declare `M_ir!AtomVar{'v}` **(displayed as** "$\downarrow v$"**)**
declare `M_ir!AtomFunVar{'R; 'v}` **(displayed as** "$R.v$"**)**

A.2.2. *Expressions*

General expressions are not values. There are several simple kinds of expressions, for conditionals, allocation, function calls, and array operations.

declare `M_ir!LetAtom{'a; v. 'e['v]}`
  (displayed as "let $v = a$ **in** $e[v]$")
declare `M_ir!If{'a; 'e1; 'e2}`
  (displayed as "if a **then** e_1 **else** e_2")
declare `M_ir!ArgNil` **(displayed as "")**
declare `M_ir!ArgCons{'a; 'rest}`
  (displayed as "a :: *rest*")
declare `M_ir!TailCall{'f; 'args}`
  (displayed as "tailcall f *args*")
declare `M_ir!Length[i:n]` **(displayed as** "i")
declare `M_ir!AllocTupleNil` **(displayed as "()")**
declare `M_ir!AllocTupleCons{'a; 'rest}`
  (displayed as "(a :: *rest*)")
declare `M_ir!LetTuple{'length; 'tuple; v. 'e['v]}`
  (displayed as
  "let v =[**length** = *length*] *tuple* **in** $e[v]$")
declare `M_ir!LetSubscript{'a1; 'a2; v. 'e['v]}`
  (displayed as "let $v = a_1.[a_2]$ **in** $e[v]$")
declare `M_ir!SetSubscript{'a1; 'a2; 'a3; 'e}`
  (displayed as "$a_1.[a_2] \leftarrow a_3;\ e$")

Reserve statements are used to specify how much memory may be allocated in a function body. The M_reserve module defines an explicit phase that calculates memory usage and adds reserve statements. In the **reserve** *words* **words args** *args* **in** e expressions, the *words* constant defines how much memory is to be reserved; the *args* defines the set of live variables (this information is used by the garbage collector), and e is the nested expression that performs the allocation.

declare `M_ir!Reserve[words:n]{'e}`
  (displayed as "reserve *words* **words in** e")
declare `M_ir!Reserve[words:n]{'args; 'e}`
  (displayed as
  "reserve *words* **words args** *args* **in**
  e")
declare `M_ir!ReserveCons{'a; 'rest}`
  (displayed as "$ReserveCons\{a;\ rest\}$")
declare `M_ir!ReserveNil` **(displayed as "")**

`LetApply`, `Return` are eliminated during CPS conversion. `LetClosure` is like `LetApply`, but it represents a partial application.

declare `M_ir!LetApply{'f; 'a; v. 'e['v]}`
(displayed as "let apply $v = f(a)$ **in** $e[v]$**")**
declare `M_ir!LetClosure{'a1; 'a2; f. 'e['f]}`
(displayed as
"let closure $f = a_1(a_2)$ **in** $e[f]$**")**
declare `M_ir!Return{'a}` **(displayed as "return**(a)**")**

A.2.3. *Recursive values*

We need some way to represent mutually recursive functions. The normal way to do this is to define a single recursive function, and use a switch to split the different parts. For this purpose, we define a fixpoint over a record of functions. For example, suppose we define two mutually recursive functions f and g:

```
let r2 = fix{r1. record{
                  field["f"]{lambda{x. (r1.g)(x)}};
                  field["g"]{lambda{x. (r1.f)(x)}}}}
in
   r2.f(1)
```

declare `M_ir!LetRec{R1. 'e1['R1]; R2. 'e2['R2]}`
(displayed as
"let rec R_1. $e_1[R_1]$
R_2.**in**
$e_2[R_2]$**")**

The following terms define the set of tagged fields used in the record definition. We require that all the fields be functions.

The record construction is recursive. The `Label` term is used for field tags; the `FunDef` defines a new field in the record; and the `EndDef` term terminates the record fields.

declare `M_ir!Fields{'fields}` **(displayed as "{** *fields* **}")**
declare `M_ir!Label[tag:s]` **(displayed as ""***tag***"")**
declare `M_ir!FunDef{'label; 'exp; 'rest}`
(displayed as "fun *label* = *exp rest***")**
declare `M_ir!EndDef` **(displayed as "")**

To simplify the presentation, we usually project the record fields before each of the field branches so that we can treat functions as if they were variables.

declare `M_ir!LetFun{'R; 'label; f. 'e['f]}`
(displayed as "let fun $f\ =\ R.label$ **in** $e[f]$**")**

Include a term representing initialization code.

declare `M_ir!Initialize{'e}`
(displayed as "initialization e **end")**

A.2.4. *Program sequent representation*
Programs are represented as sequents: ⟨*declarations*⟩ ; ⟨*definitions*⟩ $\vdash_m$ **exp**

For now the language is untyped, so each declaration has the form `v = exp`. A definition is an equality judgment.

declare `M_ir!exp` **(displayed as "exp")**
declare `M_ir!def{'v; 'e}` **(displayed as "**$v = e$**")**
declare `M_ir!compilable{'e}`
(displayed as "compilable e **end")**

Sequent tag for the M language.

declare sequent `M_ir!sequent_arg`
{ Term : Term ⊢ Term } : Judgment
(displayed as "$\vdash_m$**")**

A.2.5. *Subscripting.*
Tuples are listed in reverse order.

declare `M_ir!alloc_tuple{'l1; 'l2}`
(displayed as "$(alloc_tuple\{l_1\}\ ::\ l_2)$**")** : Dform
declare `M_ir!alloc_tuple{'l}`
(displayed as "$alloc_tuple\{l\}$**")** : Dform

B. M_cps module

Here we define the CPS transformation in a way that aims at making the preservation of program semantics easy to verify (see Section 5 for a discussion of advantages and disadvantages of this approach).

B.1. Parents

CPS conversion is a direct logical extension of the IR language.

Extends `M_ir`

B.2. Resources

The `cps` resource

The `cps` resource provides a generic method for defining *CPS transformation*. The `cpsC` conversion can be used to apply this evaluator.

The implementation of the `cps` resource and the `cpsC` conversion rely on tables to store the shape of redices, together with the conversions for the reduction.

B.2.1. *Application*

CPS conversion is formalized by adding `CPS` terms that represent applications. The CPS conversion is defined as a transformation that maps these applications to a term that is the result of a CPS transformation.

- **CPSRecordVar**$[R]$ represents the application of the record R to the identity function.

- **CPSFunVar**$[f]$ represents the application of the function f to the identity function.

- **CPS**$[cont$**;** $e]$ is the CPS conversion of expression e with continuation $cont$. The interpretation is as the application $cont\ e$.

- **CPS**$[cont$**.** $fields[cont]]$ is the CPS conversion of a record body. We think of a record $\{f_1 = e_1; ...; f_n = e_n\}$ as a function from labels to expressions (on label f_i, the function returns e_i). The CPS form is $\lambda l.\lambda c.$**CPS**$[c$**;** $fields[l]]$.

- **CPS**$[a]$ is the conversion of the atom expression a (which should be the same as a, unless a includes function variables).

declare `M_cps!CPSRecordVar{'R}`
    (displayed as "CPSRecordVar[R]**")**
declare `M_cps!CPSFunVar{'f}` **(displayed as "CPSFunVar**[f]**")**
declare `M_cps!CPS{'cont; 'e}`
    (displayed as "CPS[$cont$; e]**")**
declare `M_cps!CPS{cont. 'fields['cont]}`
    (displayed as "CPS[$cont$**.** $fields$[$cont$]]**")**
declare `M_cps!CPS{'a}` **(displayed as "CPS**[a]**")**

B.2.2. *Formalizing CPS conversion*
CPS conversion is specified as a transformation of function application. Each rewrite in the transformation preserves the operational semantics of the program.

For atoms, the transformation is a no-op unless the atom is a function variable. If so, the function must be partially applied.

`!`[] **rewrite** `cps_atom_true` {| cps |} : **CPS**[$true$] $\longleftrightarrow$ $true$
`!`[] **rewrite** `cps_atom_false` {| cps |} : **CPS**[$false$] $\longleftrightarrow$ $false$
`!`[] **rewrite** `cps_atom_int` {| cps |} : **CPS**[$\#i$] $\longleftrightarrow$ $\#i$
`!`[] **rewrite** `cps_atom_var` {| cps |} : **CPS**[$\downarrow v$] $\longleftrightarrow$ $\downarrow v$
`!`[] **rewrite** `cps_atom_binop` {| cps |} :
  CPS[a_1 op a_2] $\longleftrightarrow$ **CPS**[a_1] op **CPS**[a_2]
`!`[] **rewrite** `cps_atom_relop` {| cps |} :
  CPS[a_1 op a_2] $\longleftrightarrow$ **CPS**[a_1] op **CPS**[a_2]
`!`[] **rewrite** `cps_fun_var` {| cps |} : **CPS[CPSFunVar**[f]] $\longleftrightarrow$ $\downarrow f$
`!`[] **rewrite** `cps_alloc_tuple_nil` {| cps |} : **CPS[()]** $\longleftrightarrow$ **()**
`!`[] **rewrite** `cps_alloc_tuple_cons` {| cps |} :
  CPS[(a :: $rest$)] $\longleftrightarrow$ (**CPS**[a] :: **CPS**[$rest$])
`!`[] **rewrite** `cps_arg_cons` {| cps |} :
  CPS[a :: $rest$] $\longleftrightarrow$ **CPS**[a] :: **CPS**[$rest$]
`!`[] **rewrite** `cps_arg_nil` {| cps |} : **CPS**[] $\longleftrightarrow$
`!`[] **rewrite** `cps_length` {| cps |} : **CPS**[i] $\longleftrightarrow$ i

CPS transformation for expressions. In the following cases, the transformation is defined by the CPS conversion of the subterms. In other words, CPS conversion commutes with the following terms.

`!`[] **rewrite** `cps_let_atom` {| cps |} :
  CPS[$cont$; **let** v = a **in** $e[v]$]
    $\longleftrightarrow$
  let v = **CPS**[a] **in CPS**[$cont$; $e[v]$]

$![]$ **rewrite** `cps_let_tuple` $\{|$ cps $|\}$:
 $\mathbf{CPS}[cont;\ \mathbf{let}\ v = [\mathbf{length} = length]\ tuple\ \mathbf{in}\ e[v]]$
 $\longleftrightarrow$
 $\mathbf{let}\ v = [\mathbf{length} = \mathbf{CPS}[length]]\ \mathbf{CPS}[tuple]\ \mathbf{in}$
 $\mathbf{CPS}[cont;\ e[v]]$

$![]$ **rewrite** `cps_let_subscript` $\{|$ cps $|\}$:
 $\mathbf{CPS}[cont;\ \mathbf{let}\ v = a_1.[a_2]\ \mathbf{in}\ e[v]]$
 $\longleftrightarrow$
 $\mathbf{let}\ v = \mathbf{CPS}[a_1].[\mathbf{CPS}[a_2]]\ \mathbf{in}\ \mathbf{CPS}[cont;\ e[v]]$

$![]$ **rewrite** `cps_set_subscript` $\{|$ cps $|\}$:
 $\mathbf{CPS}[cont;\ a_1.[a_2] \leftarrow a_3;\ e]$
 $\longleftrightarrow$
 $\mathbf{CPS}[a_1].[\mathbf{CPS}[a_2]] \leftarrow \mathbf{CPS}[a_3];\ \mathbf{CPS}[cont;\ e]$

$![]$ **rewrite** `cps_if` $\{|$ cps $|\}$:
 $\mathbf{CPS}[cont;\ \mathbf{if}\ a\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2]$
 $\longleftrightarrow$
 $\mathbf{if}\ \mathbf{CPS}[a]\ \mathbf{then}\ \mathbf{CPS}[cont;\ e_1]\ \mathbf{else}\ \mathbf{CPS}[cont;\ e_2]$

$![]$ **rewrite** `cps_let_apply` $\{|$ cps $|\}$:
 $\mathbf{CPS}[cont;\ \mathbf{let\ apply}\ v = \mathbf{CPSFunVar}[f](a_2)\ \mathbf{in}\ e[v]]$
 $\longleftrightarrow$
 $\mathbf{let\ rec}\ R.\ \mathbf{fun}\ "g" = (\lambda_a v.\ \mathbf{CPS}[cont;\ e[v]])$
 $R.\mathbf{in}$
 $\mathbf{let\ fun}\ g = R."g"\ \mathbf{in\ tailcall}\ \downarrow f\ (\downarrow g,\ \mathbf{CPS}[a_2])$

The following rules specify CPS transformation of functions and application expressions.

$![]$ **rewrite** `cps_let_rec` $\{|$ cps $|\}$:
 $\mathbf{CPS}[cont;\ \mathbf{let\ rec}\ R_1.\ fields[R_1]\ R_2.\mathbf{in}\ e[R_2]]$
 $\longleftrightarrow$
 $\mathbf{let\ rec}\ R_1.$
 $\mathbf{CPS}[cont.\ \mathbf{CPS}[cont;\ fields[\mathbf{CPSRecordVar}[R_1]]]]$
 $R_2.\mathbf{in}$
 $\mathbf{CPS}[cont;\ e[\mathbf{CPSRecordVar}[R_2]]]$

$![]$ **rewrite** `cps_fields` $\{|$ cps $|\}$:
 $\mathbf{CPS}[cont.\ \mathbf{CPS}[cont;\ \{\ fields[cont]\ \}]]$
 $\longleftrightarrow$
 $\{\ \mathbf{CPS}[cont.\ \mathbf{CPS}[cont;\ fields[cont]]]\ \}$

$![]$ **rewrite** `cps_fun_def` $\{|$ cps $|\}$:
 $\mathbf{CPS}[cont.\ \mathbf{CPS}[cont;\ \mathbf{fun}\ label = (\lambda_a v.\ e[v])\ rest]]$
 $\longleftrightarrow$
 $\mathbf{fun}\ label = (\lambda_a cont.\ \lambda_a v.\ \mathbf{CPS}[cont;\ e[v]])$
 $\mathbf{CPS}[cont.\ \mathbf{CPS}[cont;\ rest]]$

$!\,[]$ **rewrite** `cps_end_def` {| cps |} :
  CPS[*cont*. **CPS**[*cont*;]] $\longleftrightarrow$
$!\,[]$ **rewrite** `cps_initialize` {| cps |} :
  CPS[*cont*; **initialization** *e* **end**]
    $\longleftrightarrow$
  initialization CPS[*cont*; *e*] **end**
$!\,[]$ **rewrite** `cps_let_fun` {| cps |} :
  CPS[*cont*; **let fun** *f* = **CPSRecordVar**[*R*].*label* **in** *e*[*f*]]
    $\longleftrightarrow$
  let fun *f* = *R*.*label* **in CPS**[*cont*; *e*[**CPSFunVar**[*f*]]]
$!\,[]$ **rewrite** `cps_return` {| cps |} :
  CPS[*cont*; **return**(*a*)] $\longleftrightarrow$ **tailcall** $\downarrow$ *cont* (**CPS**[*a*])
$!\,[]$ **rewrite** `cps_tailcall` {| cps |} :
  CPS[*cont*; **tailcall CPSFunVar**[*f*] *args*]
    $\longleftrightarrow$
  tailcall $\downarrow f$ ($\downarrow$ *cont* :: **CPS**[*args*])
$!\,[]$ **rewrite** `cps_fun_var_cleanup` {| cps |} :
  $\downarrow$ **CPSFunVar**[*f*] $\longleftrightarrow$ **CPSFunVar**[*f*]

CPS conversion is specified as a proof rule: a program is "compilable" if the CPS conversion of the program is also compilable.

$!\,[]$ **rule** `cps_prog` :

  1. $\langle\Gamma\rangle$
  2. *cont* : **exp**
  $\vdash_m$
    compilable
      let rec *R*. **fun** ".*init*" = (λ_a*cont*. **CPS**[*cont*; *e*])
      R.**in**
      let fun *init* = *R*.".*init*" **in**
        initialization tailcall $\downarrow$ *init* ($\downarrow$ *cont*) **end**
    end $\longrightarrow$
  $\langle\Gamma\rangle \vdash_m$ **compilable** *e* **end**