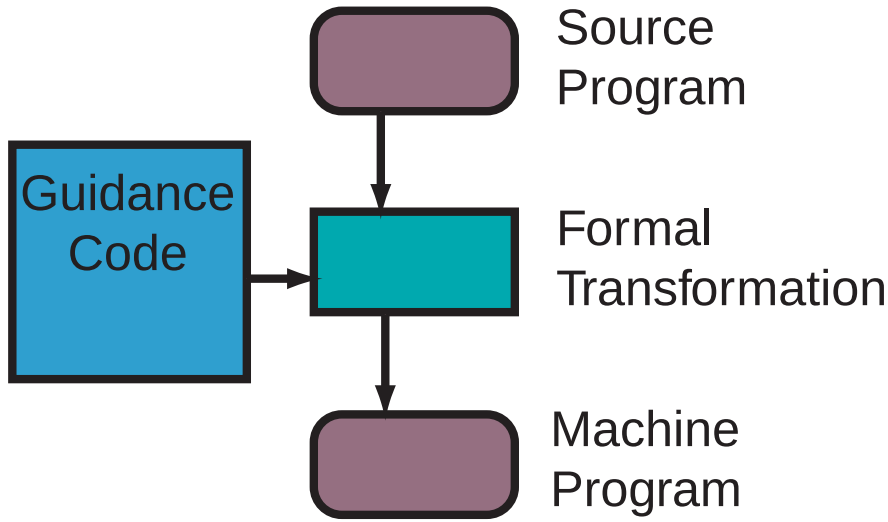
Formal Compiler Implementation in a Logical Framework

Jason Hickey, Aleksey Nogin, Adam Granicz, Brian Aydemir

Caltech Computer Science

What is a formal compiler?

- Program transformations are specified as rewrite rules
- These rules are guided by *tactics* and *conversionals* (rewrite tactics)
- Partial correctness depends only on the formal part



The MetaPRL Logical framework

- The framework provides
  - *A syntax of* terms *with binding*
  - *Substitution, alpha-equality are built-in*
  - *An LCF-style tactic mechanism to automate proofs (and compiling)*
- Programs are represented using higher-order abstract syntax
  - *Program variables are meta-variables*
  - *The framework guarantees that binding is preserved*

Advantages of using a logical framework

- The formal rules can be *verified*
  - *The formal part of the compiler is just 100s of lines of code in mathematical notation*
  - *The guidance code need not be trusted*
- The use of HOAS means that scoping and substitution are managed by the logical framework
- It is often easier to write a compiler in a LF
  - *The terminology matches that in the literature*
  - *The framework provides a great deal of automation and rewriting strategies*
  - *It is easy to experiment*

Outline of the case study

- Syntax of the language (a small untyped ML-like language)
- Compiling phases (parsing, CPS, closure conversion, code generation)
  - *Functional assembly code*
- Summary and comparison with related work

Syntax

- Generic terms in MetaPRL
  - *Each term has a name,some parameters, and some subterms with possible binding occurrences*

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

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

Term rewrites

- Rewrites are specified using second-order notation

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

  - $v_1[x]$ is a second-order pattern that specifies an arbitrary term where x may be free
  - $v_1[v_2]$ substitutes the term matched by v_2 for x in the term matched by v_1

- The following rule is also valid

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

Phases of the compiler

- Parsing
  - *Use a PDA with a term rewriting system to translate source text into terms*
- CPS
  - *An optional phase to translate into continuation-passing style*
- Closure conversion
  - *All functions become closed, top-level*
- Code generation
  - *Translate to functional x86 assembly code*
- Register allocation
  - *Convert to using x86 register set (spilling + alpha conversion)*

Parsing

- Parser is specified with a lexical analyzer, and a LR(1) parser where semantic actions are rewrite rules
- A lexical definition

$$\texttt{NUM} = \texttt{"[0 - 9] + "} \qquad \mathbf{token}[i:s]\{pos\} \longleftrightarrow number[i:n]$$

- A production

$$\texttt{exp} ::= \texttt{LET ID}\ \langle \texttt{v} \rangle\ \texttt{EQ exp}\ \langle \texttt{e} \rangle\ \texttt{IN exp}\ \langle rest \rangle$$
$$\longleftrightarrow \mathbf{let}\ v = e\ \mathbf{in}\ rest$$

IR syntax

bop	::=	$+ \mid - \mid * \mid /$	
rop	::=	$\leq \mid < \mid >$	
	$\mid$	$\geq \mid = \mid \neq$	
l	::=	*string*	
a	::=	$\top \mid \bot \mid i \mid v \mid a_1\ bop\ a_2$	
e	::=	**let** $v = a$ **in** e	Variable definition
	$\mid$	**if** a **then** e_1 **else** e_2	Conditional
	$\mid$	**let** $v = (a_1, \ldots, a_n)$ **in** e	Tuple allocation
	$\mid$	**let** $v = a_1.[a_2]$ **in** e	Subscripting
	$\mid$	$a_1.[a_2] \leftarrow a_3; e$	Assignment
	$\mid$	**let** $v = a(a_1, \ldots, a_n)$ **in** e	Function application
	$\mid$	**let** $v = a_1(a_2)$ **in** e	Closure creation
	$\mid$	**return**(a)	Return a value
	$\mid$	$a(a_1, \ldots, a_n)$	Tail-call
	$\mid$	**let rec** $R.d$ **in** e	Recursive functions
e_λ	::=	$\lambda v.e_\lambda \mid \lambda v.e$	Functions
d	::=	**let** $l = e_\lambda$ **and** d	Function definitions
	$\mid$	**end**	

CPS conversion

- A program is compilable if it is a sequence of valid assembly instructions
- A program is compilable if its CPS translation is:

$$\frac{\Gamma, c : exp \vdash \textbf{compilable}\ CPS_c(e)}{\Gamma \vdash \textbf{compilable}\ e}$$

- The term $CPS_c(e)$ represrents the application of some function c to the program e

CPS conversion in three parts

- Replace $f = \lambda x.e[x]$ with $\lambda c.\lambda x.CPS_c(e[x])$ and replace f with the partial application $f[\mathbf{id}]$, where **id** is the identity function.
- Replace tail-calls $CPS_c(f[\mathbf{id}](a_1, \ldots, a_n))$ with $f(c, a_1, \ldots, a_n)$, and $CPS_c(\mathbf{return}(a))$ with $c(a)$
- Finally, replace inline-calls $CPS_c(\mathbf{let}\, v = f[\mathbf{id}](a_1, \ldots, a_n)\ \mathbf{in}\ e)$ with $\mathbf{let}\, g = \lambda v.CPS_c(e)\ \mathbf{in}\ f(g, a_1, \ldots, a_n)$.

CPS example

1. $CPS_c(\textbf{let}\ f = \lambda x.$
   $\textbf{let}\ y = x + 1\ \textbf{in}$
   $\textbf{let}\ z = f(y)\ \textbf{in}$
   $\textbf{return}(z))$

2. $\textbf{let}\ f = \lambda c, x.$
   $CPS_c(\textbf{let}\ y = x + 1\ \textbf{in}$
   $\textbf{let}\ z = f[\textbf{id}](y)\ \textbf{in}$
   $\textbf{return}(z))$

3. $\textbf{let}\ f = \lambda c, x.$
   $\textbf{let}\ y = x + 1\ \textbf{in}$
   $CPS_c(\textbf{let}\ z = f[\textbf{id}](y)\ \textbf{in}$
   $\textbf{return}(z))$

4. $\textbf{let}\ f = \lambda c, x.$
   $\textbf{let}\ y = x + 1\ \textbf{in}$
   $f(c, y)$

A few CPS rewrites

$$
\begin{array}{ll}
\text{[fundef]} & CPS_c(\textbf{let}\, l = \lambda v.e[v]\ \textbf{and}\ d) \\
\longleftrightarrow & \textbf{let}\, l = \lambda c.\lambda v.CPS_c(e[v])\ \textbf{and}\ CPS_c(d) \\
\text{[atom]} & CPS_c(\textbf{let}\, v = a\ \textbf{in}\ e[v]) \\
\longleftrightarrow & \textbf{let}\, v = a\ \textbf{in}\ CPS_c(e[v]) \\
\text{[return]} & CPS_c(\textbf{return}(a)) \\
\longleftrightarrow & c(a)
\end{array}
$$

Closure conversion

- Another fairly straightforward phase
  - *Introduce an "environment" or "frame" to each function*
  - *For each function, quantify free variables, and add them to the frame*
  - *Propagate frame projections into function bodies*

Example

1. $\ldots$

$$\begin{array}{l}\textbf{let } f = \lambda c, y. \\ \qquad \textbf{let } z = x + y \textbf{ in} \\ \qquad\qquad c(z) \\ \textbf{in } f(exit, 1)\end{array}$$

2. $\ldots$

$$\begin{array}{l}\textbf{let } f = \lambda \mathbf{Fr}, c, y. \\ \qquad \textbf{let } z = x + y \textbf{ in} \\ \qquad\qquad c(z) \\ \textbf{with Fr} = () \textbf{ in} \\ f(\mathbf{Fr}, exit, 1)\end{array}$$

3. $\ldots$

$$\begin{array}{l}\textbf{let } x = x \textbf{ in} \\ \textbf{let } f = \lambda \mathbf{Fr}, c, y. \\ \qquad \textbf{let } z = x + y \textbf{ in} \\ \qquad\qquad c(z) \\ \textbf{with Fr} = () \textbf{ in} \\ f(\mathbf{Fr}, exit, 1)\end{array}$$

4. $\ldots$

$$\begin{array}{l}\textbf{let } f = \lambda \mathbf{Fr}, c, y. \\ \qquad \textbf{let } x = \mathbf{Fr}.[0] \textbf{ in} \\ \qquad \textbf{let } z = x + y \textbf{ in} \\ \qquad\qquad c(z) \\ \textbf{with Fr} = (x) \textbf{ in} \\ f(\mathbf{Fr}, exit, 1)\end{array}$$

Abstracting free variables

- Abstract free variables:

$$\text{[abs]} \quad e[v] \longleftrightarrow \textbf{let}\ v = v\ \textbf{in}\ e[v]$$

- Close by adding the var to the frame:

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

X86 code generation

- We retain the use of HOAS
- Implications:
  - *Registers are represented as variables*
  - *Variables must be immutable*

Functional x86 assembly code: operands

l	::=	*string*	Function labels
r	::=	*eax*|*ebx*|*ecx*|*edx*	Registers
	|	*esi*|*edi*|*esp*|*ebp*	
v	::=	$r \mid v_1, v_2, \ldots$	Variables
o_m	::=	$(\%v)$	Memory operands
	|	$i(\%v)$	
	|	$i_2(\%v_1, \%v_2, i_1)$	
o_r	::=	$\%v$	Register operand
o	::=	$o_m \mid o_r$	General operands
	|	$\#i$	
	|	$v.l$	Label

Function x86 assembly code: instructions

$$
\begin{array}{rcll}
p & ::= & \textbf{let}\ R = d\ \textbf{in}\ p \mid e & \text{Programs} \\
d & ::= & \textbf{let}\ l = e_\lambda\ \textbf{and}\ d \mid \textbf{end} & \text{Function definition} \\
e_\lambda & ::= & \textbf{let}\ v = e_\lambda \mid e & \text{Functions} \\
cc & ::= & = \mid <> \mid < \mid > \mid \leq \mid \geq & \\
inst1 & ::= & INC \mid DEC \mid \cdots & \\
inst2 & ::= & ADD \mid SUB \mid AND \mid \cdots & \\
inst3 & ::= & MUL \mid DIV & \\
cmp & ::= & CMP \mid TEST & \\
jmp & ::= & JMP & \\
jcc & ::= & JEQ \mid JLT \mid JGT \mid \cdots &
\end{array}
$$

$$
\begin{array}{rcl}
e & ::= & mov\ \ o, \lambda v.e \\
 & \mid & inst1\ \ o_m; e \\
 & \mid & inst1\ \ o_r, \lambda v.e \\
 & \mid & inst2\ \ o_r, o_m; e \\
 & \mid & inst2\ \ o, o_r, \lambda v.e \\
 & \mid & inst3\ \ o, o_r, \lambda o_r, v_1.v_2 e \\
 & \mid & cmp\ \ o_1, o_2; \\
 & \mid & jmp\ \ o(o_r \ldots o_r) \\
 & \mid & jcc\ \ e_1\ \textbf{else}\ e_2
\end{array}
$$

Example: a factorial program

- *a* is the accumulator
- *i* is the index
- *c* is the continuation

$$
\begin{array}{l}
\textbf{let } fact(c, a, i) = \\
\quad CMP \quad \%i, \#0 \\
\quad JEQ \\
\qquad JMP \quad c(a) \\
\quad \textbf{else} \\
\qquad MUL \quad \%a, \%i, \lambda a'. \\
\qquad SUB \quad \%i, \#1, \lambda i'. \\
\qquad JMP \quad fact(c, a', i')
\end{array}
$$

Code generation

- Very straightforward
  - *Translate each expression into a sequence of instructions*

[add] $\longleftrightarrow$

$$
\begin{aligned}
&ASM(a_1 + a_2), \lambda v.e[v] \\
&\quad ASM(a_1), \lambda v_1. \\
&\quad ASM(a_2), \lambda v_2. \\
&\quad ADD \quad v_1, v_2, \lambda tmp. \\
&\quad DEC \quad \%tmp, \lambda sum. \\
&\quad e[\%sum]
\end{aligned}
$$

Register allocation

- An alpha-equivalence problem
  - *Rename variables so that only the register names are used*
  - *Ensure the calling convention*

$$
\begin{array}{l}
\textbf{let } fact(eax, ebx, ecx) = \\
\quad CMP \quad \%ecx, \#0 \\
\quad JEQ \\
\quad\quad JMP \quad eax(\%ebx) \\
\quad \textbf{else} \\
\quad\quad MUL \quad \%ebx, \%ecx, \lambda ebx. \\
\quad\quad SUB \quad \%ecx, \#1, \lambda ecx. \\
\quad\quad JMP \quad fact(eax, ebx, ecx)
\end{array}
$$

Register spilling

- Suppose the register allocator is not able to assign registers, and determines that some variable *v* should be spilled
  - *We don't want to spill <u>every</u> occurrence of* v

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

Summary

- Compiler implementation in a logical framework is *easy*
- *Size*
  - *~400 lines of formal rewrites*
  - *~10k lines OCaml code (mostly in the global register allocator)*
- The case study took about 1 person/week to implement

Related work

- Liang, Compiler implementation on -Prolog, 2002
  - *Validation is not a concern yet*
  - *Everything is done in -Prolog, while we separate formal and informal parts*
- Shinwell, Fresh O'Caml, Mer yn 2003
  - *Addresses the problem of programming with binders*