Compiler Construction in Formal Logical Frameworks

Jason Hickey
Caltech

Oregon Summer School on Logic and Theorem Proving in Programming Languages
**Links**

- **MetaPRL**: [http://www.metaprl.org](http://www.metaprl.org)
- **OMake**
  - `svn co svn://svn.metaprl.org/omake-branches/jumbo/everything`
- **MetaPRL**
  - `svn co svn://svn.metaprl.org/metaprl-branches/ocaml-3.10.0`
- **Compiler**
  - `svn co svn://svn.metaprl.org/mpcompiler`
Compiler (highly simplified)

Compiler

\[ L_{\text{in}} \rightarrow L_{\text{out}} \]

\[ p_1 : \text{ML} \quad \rightarrow \quad p_2 : \text{x86} \]

Requirement: \( p_1 = p_2 \)
Logical Framework (highly simplified)

• Concepts:
  – Judgments, inferences, proofs, program extraction, etc.

• Techniques
  – Refinement, term rewriting, tactics, search, etc.

• LCF:
  – Informal tactics in ML for proof automation
  – Proofs are foundational

Logic definition + Proof automation

Meta-logic + Inference engine
Plan

- Given $p_1$, use **term rewriting** to find $p_2$ s.t. $p_1 = p_2$
- (for some $p_1$, there exists $p_2$. $p_1 = p_2$)
- NB: we will discuss certification, but we will focus primarily on program transformation

Proof

\[ p_1 \xrightarrow{\text{rewrite}} p_2 \]
Why?

- LFs provide a rich toolbox for manipulating programs
  - Transformations use textbook-style definitions
  - Transformations are cleanly isolated and defined
  - Basic concepts like alpha-renaming and substitution are builtin and automatically enforced (capture is impossible)

- Compiler is easier to write, cleanly defined, and smaller

- However: non-local transformations might be harder
  - e.g. global code motion
Outline

• Formal part: concise and precise
• Automation:
  – *usually small, sometimes not* (e.g. register allocation)
  – *LCF-style: correctness does not depend on automation*
What’s covered

• Techniques
  – Methods, representations, etc.
• Assumes
  – Some knowledge of PL + higher-order logics
  – Some knowledge of compilation
• Mostly not covered
  – Compiler verification
  – Automation
Credit

- Brian Aydemir’s undergraduate research project

- Aleksey Nogin, Nathan Gray, ...

http://www.cs.uoregon.edu/research/summerschool/summer08/
**Concerns**

- **Real** compilers have many stages and many representations

- Compositionality is a fundamental concern
Outline

• Logical frameworks
  – Concepts and tools
• Compilers
  – Methods and stages
• Compiler implementation in a LF
Logical Frameworks

• A logical framework is a formal meta-language for deductive systems [Pfenning]; it allows
  – specification of deductive systems,
  – search for derivations within deductive systems,
  – meta-programming of algorithms pertaining to deductive systems,
  – proving meta-theorems about deductive systems.

• Some Logical Framework systems: ELF, Twelf, Isabelle, lambda-Prolog, MetaPRL.
Logical framework

- A language (and a syntax)
- Inferences and derivations
- Search
Explicit term syntax.

\[ t ::= \text{opname}[p_1, \ldots, p_n] \{b_1; \ldots; b_m\} \quad \text{terms} \]
\[ | \quad x, y, z, \ldots \quad \text{variables} \]

\[ p ::= 0, 1, 2, \ldots \quad \text{parameters (constants)} \]
\[ | \quad "aaa", \ldots \quad \text{string constants} \]

\[ b ::= x_1, \ldots, x_n.t \quad \text{bound term} \]
**Syntax**

- **Binders** are primitive (not functions).
- Convention: omit empty parameter, binder, and bterm lists.
- Examples:

<table>
<thead>
<tr>
<th>Pretty form</th>
<th>Actual syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>number[1]{}</td>
</tr>
<tr>
<td>1 + 2</td>
<td>add{number[1]; number[2]}</td>
</tr>
<tr>
<td>(\lambda x.x)</td>
<td>lambda{x.x}</td>
</tr>
<tr>
<td>((\lambda x.x) \ 1)</td>
<td>apply{lambda{x. x}; number[1]}</td>
</tr>
</tbody>
</table>
Patterns and schemas

- Patterns are specified with *second-order* variables.

\[
t ::= \cdots \quad \text{terms} \\
| \ x[y_1; \cdots; y_n]
\]

- The so-variable \( x[y_1; \cdots; y_n] \) *stands for* an arbitrary term, where the only free variables are \( y_1, \ldots, y_n \).

- The so-variable \( x[\] \) stands for an arbitrary closed term.
Matching

- A second-order variable matches any term, with constraints on free variables

- (Using the usual $\alpha$-renaming convention)

<table>
<thead>
<tr>
<th>Term</th>
<th>Pattern</th>
<th>Match</th>
</tr>
</thead>
<tbody>
<tr>
<td>$y + y$</td>
<td>$x[y]$</td>
<td>$x[y] = y + y$</td>
</tr>
<tr>
<td>$y + z$</td>
<td>$x[y]$</td>
<td>no match</td>
</tr>
<tr>
<td>$1 + 2$</td>
<td>$x[y]$</td>
<td>$x[y] = 1 + 2$</td>
</tr>
<tr>
<td>$\lambda z.z + z$</td>
<td>$\lambda y.x[y]$</td>
<td>$x[y] = y + y$</td>
</tr>
</tbody>
</table>
Substitution

• Given a matching

\[ x[y_1; \cdots; y_n] = t \]

a so-term \( x[s_1; \cdots; s_n] \) is a substitution

\[ x[s_1; \cdots; s_n] \equiv t[s_1/y_1; \cdots; s_n/y_n] \]
Term rewriting specifications

- Definition of $\beta$-equivalence:

  $$(\lambda x.e_1[x])\ e_2[] \leftrightarrow e_1[e_2[]]$$

  ($e_1[x]$ and $e_2[]$ are second-order).

- Rewrite application:

  $$(\lambda y.y + 1)\ 2 \leftrightarrow 2 + 1$$

- where,

  $$e_1[x] = x + 1$$
  $$e_2[] = 2$$
  $$e_1[e_2[]] = 2 + 1$$
**Contexts**

- **Contexts** $\Gamma[x]$ are like so-variables, but they represent a term with a single hole

- Contexts are frequently used in sequent terms

- $\Gamma[x : t[]; \Delta[\vdash x \in t[]]]$

- Pretty form:  
  $$\Gamma; x : t ; \Delta \vdash x \in t$$
Sentences

- Sentences in the meta-logic are called *schemas*, second-order Horn-formulas, of the form

\[ t_1 \rightarrow t_2 \rightarrow \cdots \rightarrow t_n \]

- usually written like an inference rule

\[
\begin{array}{c}
t_1 \\
t_2 \\
\vdots \\
t_{n-1}
\end{array}
\begin{array}{c}
\rightarrow \\
\rightarrow \\
\rightarrow \\
\rightarrow
\end{array}
\begin{array}{c}
t_n
\end{array}
\]

- **closed** w.r.t. first-order variables

- so variables are implicitly *universally* quantified
Inference in the meta-logic

- The only meta-logical inference rule is refinement (like resolution).

- It is exactly what you expect!

  - Suppose $t_1 \rightarrow \cdots \rightarrow t_n \rightarrow u$ is an axiom.
  - To prove $s_1 \rightarrow \cdots \rightarrow s_m \rightarrow u$
  - You must prove $s_1 \rightarrow \cdots \rightarrow s_m \rightarrow t_i$
    for each $1 \leq i \leq n$. 
Logics

• Defining and using a logic:
  – Declare some terms that specify the syntax of formulas in your logic,
  – Declare some axioms for its rules,
  – (Define some proof automation),
  – Derive some facts.
Example: ST lambda calculus syntax

- Terms:
  - application: `apply{e1; e2}; pretty e_1 e_2`
  - abstraction: `lambda{t; x. e}; pretty \lambda x: t.e`
  - arrow type: `fun{t1; t2}; pretty t_1 \rightarrow t_2`
  - type judgment: `mem{e; t}; pretty e : t`
  - judgment: `\Gamma \vdash e : t`; (not writing ugly form!)
Axioms for static semantics

\[
\Gamma; x : t; \Delta \vdash x : t \quad \text{var}
\]

\[
\Gamma \vdash e_1 : s \rightarrow t \quad \Gamma \vdash e_2 : s \\
\Gamma \vdash e_1 \ e_2 : t \quad \text{app}
\]

\[
\Gamma, x : s \vdash e : t \\
\Gamma \vdash (\lambda x : s.e) : s \rightarrow t \quad \text{abs}
\]

- Context variables: \( \Gamma \)
- Second-order variables: \( s, t, e, e_1, e_2 \)
- First-order variables: \( x \)
Rewrites

- Term rewriting is just a special case of a rule.
- A rewrite definition

\[
\begin{align*}
  s_1 & \xrightarrow{\text{\cdots}} s_n \xrightarrow{(t_1 \leftrightarrow t_2)} \\
  s_1 & \xrightarrow{\text{\cdots}} s_n \xrightarrow{\Gamma[t_1]} \xrightarrow{\Gamma[t_2]} \\
  s_1 & \xrightarrow{\text{\cdots}} s_n \xrightarrow{\Gamma[t_2]} \xrightarrow{\Gamma[t_1]}
\end{align*}
\]

means \( t_1 \) and \( t_2 \) are equivalent in any context.
Dynamic semantics

- Rewriting axiom:

\[(\lambda x: t.e_1[x]) e_2 \leftrightarrow e_1[e_2]\]

- Note that \(t\) is lost by rewriting.

- This is not exactly faithful, because the rewrite is reversible.
Summary: MetaPRL LF

- Syntax
  - terms, constants, binders,
  - first-order, second-order, and context variables
- Meta-implications (inference rules)

\[
\frac{s_1 \cdots s_n \text{ foo}}{t} \quad \frac{\Gamma \vdash e_1 : S \to T \quad \Gamma \vdash e_2 : S}{\Gamma \vdash e_1 \, e_2 : T} \quad \text{app}
\]

- Meta-rewrites

\[
s \leftrightarrow t \quad (\lambda x : S.\ e_1 [x])\ e_2 \leftrightarrow e_1 [e_2]
\]
Notes

- Strictly speaking, context variables are *binders* and so-variables must specify them.

\[
\Gamma \vdash e_1[\Gamma] : S[\Gamma] \rightarrow T[\Gamma] \quad \Gamma \vdash e_2[\Gamma] : S[\Gamma] \\
\Gamma \vdash e_1[\Gamma] \; e_2[\Gamma] : T[\Gamma]
\]

\[
\Delta[(\lambda x:S[\Delta].e_1[x,\Delta]) \; e_2[\Delta]] \leftrightarrow \Delta[e_1[e_2[\Delta],\Delta]]
\]

- There is a syntactic type system that enforces syntactical well-formedness

  - In \(\lambda x:t.e[x]\), \(t\) represents a type, and \(e[x]\) represents an expression
**Task: build a compiler**

- **Char stream**
  - Lexing
- **Token stream**
  - Parsing
- **AST**
  - Semantic Analysis
- **IR**
  - Optimize
    - Constant Folding
    - Common Subexpression Elim
    - Function inlining
    - Hoisting
  - Instruction selection
    - Assem
  - Register allocation
    - Assem
  - Code emission
    - Object
Lexing, parsing, printing

- MetaPRL includes parsers+printers
  - defined together with the logic
  - standard technology LALR(1), boring

\[
\text{fun (x : t) -> e} \\
\text{LALR(1) Extensible parsers}
\]

\[
\text{fun (x: t) -> e} \\
\text{Pretty-printer (rewrite based)}
\]

\[
\text{Concrete terms}
\]

\[
\text{lambda\{t; x.e\}}
\]
Actual plan

Char stream
  | Lexing
Token stream
  | Parsing
AST
  | Semantic Analysis

IR
  | Optimize

Lexing
Parsing
Semantic Analysis
Optimize
Constant Folding
Common Subexpression Elim
Function inlining
Hoisting

Instruction selection
Register allocation
Code emission

Char stream
Token stream
AST
IR
Object
Part I: Syntax

• Most mainstream compilers are monolithic w.r.t. the source language
• But we want languages to be extensible, just like a logic
  – Start with a core language
  – Add extensions to it later
Core language: ML-like

\[ e ::= \]
\[ | \quad e(e_1, \ldots, e_n) \quad \text{expressions} \]
\[ | \quad \text{fun}(x_1, \ldots, x_m) \to e \quad \text{application} \]
\[ | \quad \text{let } x = e_1 \text{ in } e_2 \quad \text{function} \]
\[ | \quad \text{let rec } x_1 = e_1 \text{ and } \cdots \text{ and } x_n = e_n \text{ in } e \quad \text{let} \]
\[ | \quad \text{let rec } x_1 = e_1 \text{ and } \cdots \text{ and } x_n = e_n \text{ in } e \quad \text{recursive definition} \]

• Notes:

- Arbitrary arity is achieved using *sequents*

\[ \text{fun}(x_1, \ldots, x_n) \to e \equiv x_1:_-, \ldots, x_n:_- \vdash \lambda e \]
\[ e(e_1, \ldots, e_n) \equiv e(_:e_1, \ldots, _:e_n \vdash \text{args } _) \]

- Variables (first-order, second-order, context) are implicitly included in the language.
Compiler judgment

- Primary judgment $\langle\langle e \rangle\rangle$
  - Pronounced “$e$ is compilable”
  - Intent: $e$ is compilable iff there is a machine program $e'$ equivalent to it.

- To compile a program $p$
  - Constructively prove a theorem $\vdash \langle\langle p \rangle\rangle$
  - The *witness* machine program $p'$ is the result
Compilable

- This is a \textit{partial} argument
  - \textit{The proof may fail because}
    - our compiler is incorrectly automated
    - doesn't terminate
    - the source program is “incorrect”
  - \textit{Translation validation: if a proof is found, it is correct}

- First step: how do we prove \texttt{<<e>>}?
Part II: types and type inference

- We’ll use a typed intermediate language.
- Define a type erasure function $\text{erase}$,
- and a typed $\langle \langle e : t \rangle \rangle$ “compilable” judgment.

\[
\begin{align*}
\Gamma & \vdash \langle \langle e : t \rangle \rangle \\
\Gamma & \vdash \langle \langle \text{erase}(e) \rangle \rangle \quad \text{infer}
\end{align*}
\]

- automation: to compile an untyped program $e$,
  - find a typed program $e'$ and a type $t$
    s.t. $e = \text{erase}(e')$ and $e' : t$. 
Syntax: System F

\[ t ::= \bot \mid \top \mid \text{Fun}(t_1, \ldots, t_n) \to t \mid \forall (X_1, \ldots, X_n).t \]

base types
function type
polymorphic type

\[ e ::= e(e_1 : t_1, \ldots, e_n : t_n) \mid \text{fun}(x_1 : t_1, \ldots, x_n : t_n) \to e \mid \text{let } x : t = e_1 \text{ in } e_2 \mid \text{let rec } x_1 : t_1 = e_1 \text{ and } \cdots \text{ and } x_n : t_n = e_n \text{ in } e \mid \text{Lam}(X_1, \ldots, X_n) \to e \mid e[|t_1; \cdots; t_n|] \]

application
function
let
recursive definition
type function
type application
Type erasure

- type erasure is a *rewriting* operation
  - defined by structural induction (syntax directed)
  - some definitions are easy

\[ \text{erase}(\text{let } x : t = e_1 \text{ in } e_2) \rightarrow \text{let } x = \text{erase}(e_1) \text{ in erase}(e_2) \]

- However, rewrites can specify only a fixed number of operations
  - terms with unbounded arities must be transformed one part at a time

\[ \text{erase}(\text{fun}(x_1 : t_1, \ldots, x_n : t_n) \rightarrow e) \rightarrow \text{???} \]
Inductive definitions

- Introduce a temporary context $\Gamma \vdash \cdots$, then specify the transformation by induction in 3 parts

  - $\textit{erase}(\texttt{fun} (\Delta) \rightarrow e) \rightarrow \textit{erase}(\vdash \texttt{fun} (\Delta) \rightarrow e)$

  - $\textit{erase}(\Gamma \vdash \texttt{fun} (x_i : t_i, \Delta) \rightarrow e) \rightarrow \textit{erase}(\Gamma, x_i : \_ \vdash \texttt{fun} (\Delta) \rightarrow e)$

  - $\textit{erase}(\Gamma \vdash \texttt{fun} () \rightarrow e) \rightarrow (\texttt{fun} (\Gamma) \rightarrow \textit{erase}(e))$
Notes

• The style is similar for the other expressions
• Type erasure is syntax-directed, so:
  – it is entirely automated
  – without requiring any help from the programmer
Type checking

• Theorem provers are really good at this
• Simple fixed rules

\[
\Gamma \vdash e_1 : t \quad \Gamma, x : t \vdash e_2[x] : s \\
\hline
\Gamma \vdash \text{let } x : t = e_1 \ \text{in} \ e_2[x] : s
\]

let
Type checking unbounded arity

\[
\begin{align*}
\frac{\Gamma \vdash e : t}{\Gamma \vdash (\texttt{fun}() \to e) : (\texttt{Fun}() \to t)} & \quad \text{fun0}
\end{align*}
\]

\[
\begin{align*}
\frac{\Gamma, x : s \vdash (\texttt{Fun}(\Delta_1) \to e) : (\texttt{Fun}(\Delta_2) \to t)}{\Gamma \vdash (\texttt{fun}(x : s, \Delta_1) \to e) : (\texttt{Fun}(s, \Delta_2) \to t)} & \quad \text{fun1}
\end{align*}
\]
Type checking

- Similar structure for application, type application, etc.
- Syntax directed, fully automated

- N.B. the following rule is not valid if there are side-effects

\[
\begin{align*}
\Gamma, X \vdash (\text{Lam}(\Delta_1) \rightarrow e) : (\forall (\Delta_2) \rightarrow t) \\
\Gamma \vdash (\text{Lam}(X, \Delta_1) \rightarrow e) : (\forall (X, \Delta_2) \rightarrow t)
\end{align*}
\] Lam1
Type inference

- We have defined $erase(e)$,
- and a type judgment $\Gamma \vdash e : t$,
- and the inference,

$$\Gamma \vdash \langle \langle e : t \rangle \rangle \quad \Gamma \vdash \langle \langle erase(e) \rangle \rangle$$

- How do we find $t$?
Hindley-Milner type inference

• Given an untyped program $e$, compute $e'$ and $t$ the usual way (algorithm W), s.t. $\text{erase}(e') = e$ and $\vdash e' : t$.

• This is an example where the computation is performed outside the meta-logic

• The system still provides support, need about 300 lines OCaml code for the core language
Compiler outline

“ML” TAST .......................... assembly

CPS Closure OPT

type inference  

YOU ARE HERE

codegen register allocation
CPS

• Read Danvy and Filinski, *Representing Control: A Study of the CPS Transformation (1992)*
An example transformation

- Conversion to continuation passing style is a straightforward translation (from Danvy and Filinski)

\[
[\text{let } x = N \text{ in } M] = \\
\lambda c. \overline{\Theta[N]}(\lambda n. \text{let } x' = n \text{ in } \overline{\Theta[M[x \leftarrow x']]}c)
\]

- MetaPRL version uses meta-notation to represent transformation-time terms; meta-syntax and object-syntax are clearly separated.

\[
\text{CPS}\{\text{let } v_1 : t_1 = e_1 \text{ in } e_2[v_1]; t_2; v_2.c[v_2]\} \\
\leftarrow [\text{cps_let}] \rightarrow \\
\text{CPS}\{e_1; t_1; v_3.\text{let } v_1 : \text{TyCPS}\{t_1\} = v_3 \text{ in } \\
\text{CPS}\{e_2[v_1]; t_2; v_2.c[v_2]\}\}
\]
Closure conversion

• Also called lambda lifting
• Goal: every lambda-abstraction should be closed
  – Then, it can be hoisted to top-level
• Formal definition:
  – It is difficult (but not impossible) in this setting to talk about variables formally
  – HOAS: binders in the object language are represented as binders in the meta-language
    • free variables, names, substitution are implicit
  – See Hickey et.al. Hybrid deBruijn/HOAS in ICFP 2006 for another approach
Lightweight closure conversion

- Use term rewriting to
  - step 1: close
  - step 2: hoist
- Potential issue
  - Rewriting is local, is this possible?
Closure Conversion in 4 parts

0. Function with a free var

\[
\cdots (\text{fun}(x : t) \rightarrow x + y) \cdots
\]

1. Add a dummy let for the free var (just to get it near the fun)

\[
\cdots (\text{let } y : \mathbb{Z} = y \text{ in fun}(x : \mathbb{Z}) \rightarrow x + y) \cdots
\]
2. Add an extra parameter, and apply it

\[
\cdots \ (\text{let } y : \mathbb{Z} = y \ \text{in} \ (\text{fun}(y : \mathbb{Z}, x : \mathbb{Z}) \rightarrow x + y)(y)) \cdots
\]

3. Hoist

\[
\text{let } f = \text{fun}(y : \mathbb{Z}, x : \mathbb{Z}) \rightarrow x + y \ \text{in} \\
\cdots \ (\text{let } y : \mathbb{Z} = y \ \text{in} \ f(y)) \cdots
\]
Formal definition (parts 2, 3)

2. Purely local definition

\[
\text{let } x : t = e_1 \text{ in } \text{fun}(\Delta) \rightarrow e_2[x] \\
\leftrightarrow \text{let } x : t = e_1 \text{ in } (\text{fun}(x : t, \Delta) \rightarrow e[x])(x : t)
\]

3. Need a single context

\[
\text{let } f = e[] \text{ in } \Gamma[f] \leftrightarrow \Gamma[e[]]
\]

- \(\Gamma[e[]]\) is an arbitrary context containing \(e\)
- apply the rewrite in reverse
- note: \(e[]\) means that \(e\) is closed
Part 1 is harder

- The following rewrite is **wrong**!

  \[ e[x] \leftrightarrow \text{let} \ x : t = x \ \text{in} \ e[x] \]

- Two problems:
  - What is \( x \)? Supposed to be a first-order var.
  - What is \( t \)? Can it be anything?
Proper formal definition

• Every variable has a *binding* (we only consider closed programs),
  
  – Every binding has a type constraint (by luck?)

• Use a context to place the let-binder.

\[
\text{let } x : t = e_1 \ \text{in} \ \Gamma[e[x]] \\
\iff \\
\text{let } x : t = x_1 \ \text{in} \\
\Gamma[\text{let } x : t = x \ \text{in} \ e[x]]
\]
Generalized form

- There are several kinds of binders
- It is frequently useful to know the types of all the bound variables in a given context
- General solution: collect an environment by scanning from the root the the leaves

\[ \text{sweep}(\Sigma \models e) \]

- where \( \Sigma \) is a set of membership terms

\[ \Sigma ::= x_1 \in t_1, \ldots, x_n \in t_n \]
**Definition**

The general form of $\text{sweep}(\Sigma \models e)$ is defined by structural induction

\[
\text{sweep}(\Sigma \models \text{let } x : t = e_1 \text{ in } e_2) \\
\iff \text{let } x : t = \text{sweep}(\Sigma \models e_1) \text{ in } \text{sweep}(\Sigma, x : t \models e_2)
\]

\[
\text{sweep}(\Sigma \models \text{fun}(\Delta) \rightarrow e) \\
\iff \text{fun}(\Delta) \rightarrow \text{sweep}(\Sigma, \Delta \models e)
\]

\[
\text{sweep}(\Sigma \models e(e_1, \ldots, e_n)) \\
\iff \text{sweep}(\Sigma \models e)(\text{sweep}(\Sigma \models e_1), \ldots, \text{sweep}(\Sigma \models e_n))
\]

\[
\text{sweep}(\Sigma \models x) \iff x
\]


**Sweep let droppings**

- **Generic** rule for adding a let-definition

\[
\text{sweep}(\Sigma_1, x \in t, \Sigma_2 \vdash e[x]) \\
\iff \text{sweep}(\Sigma_1, x \in t, \Sigma_2 \vdash \text{let } x : t = x \text{ in } e[x])
\]

- Steps in closure conversion:
  - Sweep down the term, placing appropriate let-definitions before the functions
  - Add new function parameters
  - Hoist functions (now closed)
Summary: closure conversion

- Three main steps:
  - Add let-definitions for free variables
  - Add extra function parameters
  - Hoist functions

- Next
  - Can go straight to code generation
  - But, let’s do some optimizations
Outline

“ML”  TAST  assembly

CPS  Closure  OPT

type inference  you are here  codegen register allocation
**Dead code elimination**

- Dead code: any code that does not affect the behavior of the program
- Mainly introduced during code transformation
- Syntactic approximation:
  
  \[
  \text{let } x : t = e_1 \text{ in } e_2 \rightarrow e_2
  \]

  (note \(x\) is not free in \(e_2\))

- OK to apply blindly, everywhere
- Caution: what about side-effects?

---

If this page displays slowly, try turning off the "smooth line art" option in Acrobat, under Edit-Preferences
Common subexpression elimination

- Inverse beta-reduction (if language is pure)
  \[
  \textbf{let } x : t = e_1 \textbf{ in } e_2[x] \leftarrow e_1[e_2]
  \]
  
- Apply in reverse (right-to-left)
  \[
  a \ast b + f(a \ast b) \\
  \textbf{let } x : \mathbb{Z} = a \ast b \textbf{ in } \cdots x + f(x)
  \]
Inlining

• (beta-reduction)

\[
\text{let } x: t = e_1 \text{ in } e_2[x] \rightarrow e_2[e_1]
\]

\[
(\text{fun}(x: t, \Delta_1) \rightarrow e[x])(e_1, \Delta_2) \rightarrow (\text{fun}(\Delta_1) \rightarrow e[e_1])(\Delta_2)
\]

• Example:

\[
\text{let } f = \text{fun}(x: \mathbb{Z}) \rightarrow x + x \text{ in } f(1)
\]

\[
\rightarrow (\text{fun}(x: \mathbb{Z}) \rightarrow x + x)(1)
\]

\[
\rightarrow (\text{fun}() \rightarrow 1 + 1)()
\]

\[
\rightarrow 1 + 1
\]
Partial Redundancy Elimination
Partial Redundancy Elimination

• So there, Sorin!
Homework solution

- Closure conversion for recursive functions
- Recursive definitions are defined together
  - Definitions may be nested, but it doesn’t matter
  - (Assume e₁, …, en are lambdas)

\[
\begin{align*}
\text{let rec } & f_1 : t_1 = e_1 \\
\text{and } & \cdots \\
\text{and } & f_n : t_n = e_n \\
\text{in } & e
\end{align*}
\]
**Step 1: add a let-definition (simultaneous)**

- Collect free variables

\[
\text{let } \Delta_1 = \Delta_2 \text{ in } \\
\text{let rec } f_1 : t_1 = e_1 \\
\text{and } \cdots \\
\text{and } f_n : t_n = e_n \\
\text{in } e
\]

- \(\Delta_1 = (x_1 : t_1, \ldots, x_m : t_m)\)

- \(\Delta_2 = (x_1, \ldots, x_m) = FV(e_1) \cup \cdots \cup FV(e_n)\)
Step 2: Add extra function parameters

- Use new names for actuals funs, old names for partial applications

```latex
let \Delta_1 = \Delta_2 \text{ in }
let \text{rec } f'_1 : \text{Fun}(\Delta_1) \to t_1 = \text{fun}(\Delta_1) \to e_1
\text{ and } \cdots
\text{and } f'_n : \text{Fun}(\Delta_1) \to t_n = \text{fun}(\Delta_1) \to e_n
\text{ and } f_1 : t_1 = f'_1(\Delta_2)
\text{ and } \cdots
\text{and } f_n : t_n = f'_n(\Delta_2)
\text{ in } e
```
Notes:

- This is actually done 1 function at a time
  - Close \( f_1 \) in \textbf{let rec} \( f_1, \ldots, f_n \) \textbf{in} \( \ldots \)
  - Then rotate to \textbf{let rec} \( f_2, \ldots, f_n, f'_1, f_1 \) \textbf{in} \( \ldots \)

- In a real compiler, only 1 closure is needed:
  - \( c = (f'_1, \ldots, f'_n, x_1, \ldots, x_m) \)
  - \( f_i(e, \ldots, e) = \text{apply}_i(c, e, \ldots, e) \)
  - Easy to do (but the language needs to be extended)
Pretty important optimization

- Inline closures when possible

\[
\text{let } c = f(e_1, \ldots, e_m)_c \text{ in } \Delta[c(e_{m+1}, \ldots, e_n)]
\]
\[
\rightarrow \text{let } c = f(e_1, \ldots, e_m)_c \text{ in } \Delta[f(e_1, \ldots, e_m, e_{m+1}, \ldots, e_n)]
\]
Extensibility, compositionality

- The core language is unrealistically small
- We would like arithmetic, tuples, Boolean values, assignment, …
- Architecturally, we want the language to be *compositionally*
  - choose the language features

- In a LF, this style happens frequently, as logics are constructed
  - constructive propositional -> classical propositional
  - constructive propositional -> constructive first-order -> classical first-order logic -> …
Extensibility

• Formally, it is no different in a compiler
Useful example: Tuples

- New syntax
  - (Note: MetaPRL grammars are extensible)

- Untyped language

\[ e ::= \ldots \]
\[ \mid (e_1, \ldots, e_n) \quad \text{tuple} \]
\[ \mid \text{let}(x_1, \ldots, x_n) = e_1 \quad \text{projection} \]

expressions
**Tuples: typed language**

\[
t ::= \cdots \quad \begin{array}{l}
| t_1 \times \cdots \times t_n
types \\
\end{array}
\]

\[
e ::= \cdots \quad \begin{array}{l}
| (e_1 : t_1, \ldots, e_n : t_n) \quad \text{tuple} \\
| \text{let}(x_1 : t_1, \ldots, x_n : t_n) = e_1 \text{ in } e_2 \quad \text{projection}
\end{array}
\]
Tuple: type erasure

- Erasure definition

\[
\text{erase}(e_1 : t_1, \ldots, e_n : t_n) \rightarrow (\text{erase}(e_1), \ldots, \text{erase}(e_n))
\]

\[
\text{erase}(\text{let}(x_1 : t_1, \ldots, x_n : t_n) = e_1 \text{ in } e_2) \rightarrow \text{let}(x_1, \ldots, x_n) = \text{erase}(e_1) \text{ in } \text{erase}(e_2)
\]

- Automation is still automatic (just include these rewrites).
**Tuple: type checking**

\[ \Gamma \vdash () : () \quad \text{tuple}_0 \]

\[ \begin{align*}
\Gamma &\vdash e : t \\
\Gamma &\vdash (\Delta_1) : \Delta_2 \\
\Gamma &\vdash (e : t, \Delta_1) : t \, \ast \, \Delta_2
\end{align*} \quad \text{tuple}_1 \]

\[ \begin{align*}
\Gamma &\vdash e_1 : (\Delta) \\
\Gamma, \Delta &\vdash e_2 : t \\
\Gamma &\vdash (\text{let}(\Delta) = e_1 \text{ in } e_2) : t
\end{align*} \quad \text{proj} \]
**Tuple: sweep (for closure conversion)**

\[
\text{sweep}(\Sigma \vdash (e_1 : t_1, \ldots, e_n : t_n)) \\
\rightarrow (\text{sweep}(\Sigma \vdash e_1) : t_1, \ldots, \text{sweep}(\Sigma \vdash e_n) : t_n)
\]

\[
\text{sweep}(\Sigma \vdash \text{let}(\Delta) = e_1 \text{ in } e_2) \\
\rightarrow \text{let}(\Delta) = \text{sweep}(\Sigma \vdash e_1) \text{ in } \text{sweep}(\Sigma \vdash e_1)
\]
Revisiting closure conversion

- Represent the environment as a tuple

\[
\text{let}(\Delta_1) = (\Delta_2) \text{ in } \text{fun}(\Delta_3) \rightarrow e[\Delta_1, \Delta_3] \\
\leftrightarrow \quad \text{let}(\Delta_1) = (\Delta_2) \text{ in }
\quad \text{fun}(x : \cdot, \Delta_3) \rightarrow
\quad \text{let}(\Delta_1) = x \text{ in } e[\Delta_1, \Delta_3])((\Delta_2))
\]
Outline

“ML”  TAST  - - - - - - - - - - - - - - assembly

CPS  Closure  OPT

type inference  codegen register allocation

YOU ARE HERE
Code generation

• Intermediate representation
  – Fairly high-level (ML-like)
  – Typed
  – Pure
  – Explicit binders
    • alpha-equivalence, substitution make sense

• Machine code
  – Low level
  – Imperative
  – Fixed number of registers
Back-ends

- A compiler may have several back-ends, one for each instruction set architecture (ISA)
- We’ll do Intel x86 (386)
  - Please read the Intel instruction set description during the next few slides (~1000 pages)
Oversimplified x86 architecture

Register file

- eax
- edx
- ecx
- ebx
- esi
- edi
- esp
- ebp

2-operand ALU

L1 Cache (16k)

L2 Cache (512k)

Primary RAM (8MB-64GB)
2-operando instructions

// Factorial:
// Arg in %ebx
// Result in %eax
// Destroys %edx

mov %eax, $1 // %eax <- 1

fact:
cmp %ebx, $0 // test %ebx == 0
jmp z, break // if so, exit
mul %ebx // %eax *= %ebx
dec %ebx // %ebx--
jmp fact // next iteration

exit:
Notes

• Most instructions have a normal 2-operand form
  – \texttt{ADD \textit{op1},\textit{op2}}
    • means \texttt{op1 += op2}
• Some instructions are strange
  – \texttt{MUL \textit{op1}}
    • means \texttt{(edx,eax) *= op1}
  – \texttt{SHL \textit{op1},\textit{op2}}
    • means \texttt{op1 <<= op2}
    • but \texttt{op2 must be a constant or \%cl}
**x86 is a CISC architecture**

- Lots of instructions, some very complex
  
  - *For example, looping constructs, string operations*

- We will use only a simple subset

- Most complex instructions are pretty slow
  
  - *Because compiler writers often ignore the complex parts*
  
  - *Intel wouldn’t benefit much by optimizing them*
Operands

- Instruction

\[ \text{opcode} \ \text{operand}_1, \text{operand}_2 \]

- Operand

\[ \text{operand} ::= i \quad \text{address} \]
\[ \quad | \quad $i \quad \text{integer constant} \]
\[ \quad | \quad \%r \quad \text{register} \]
\[ \quad | \quad (\%r) \quad \text{indirect} - \star r \]
\[ \quad | \quad i(\%r) \quad \text{offset} - \star (r + i) \]
\[ \quad | \quad i_1(\%r_1, \%r_2, i_2) \quad \star (r_1 + r_2*i_2 + i_1) \]
Representation

• We have two choices:
• Deep embedding where we model the real machine
  – state = registers + heap + pc + flags + …
  – an instruction is a state transformation
  – this needs to be done for proving correctness
  – straightforward, and laborious

• Alternative: shallow embedding
  – Registers are represented by variables
  – The heap is abstract

• Shallow embedding is much more interesting, perhaps more appropriate(?)
X86 instruction set

- We’ll use a simplified representation
  - *Bindings are significant*
  - *3-operand instructions*
  - *Typed assembly*

- We’ll initially assume that there are an infinite number of registers/variables
  - *Register v is valid for any variable v*
  - *Register allocation will take care of assignment to actual registers*
**Abstract instruction set**

\[ e ::= \begin{align*}
\text{let } r : t &= op \text{ in } e & \text{Load} \\
& \mid op & \leftarrow \%r; e & \text{Store} \\
& \mid \text{let } r : t = op_1 + op_2 \text{ in } e & \text{arithmetic} \\
& \mid \text{let } r : t = f(r_1, \ldots, r_n) \text{ in } e & \text{function call} \\
& \mid \text{jmp } f(r_1, \ldots, r_2) & \text{unconditional branch} \\
& \mid \text{cmp } op_1, op_2; e & \text{compare} \\
& \mid \text{if } cc \text{ then } e_1 \text{ else } e_2 & \\
& \mid \text{ret } op 
\end{align*} \]

\[ p ::= \begin{align*}
\text{let rec } f_1(r, \ldots, r) &= e_1 \\
& \text{and } f_2(r, \ldots, r) &= e_2 \\
& \ldots \\
& \text{and } f_n(r, \ldots, r) &= e_n
\end{align*} \]
Notes

- A **program** is a set of recursive definitions called **basic blocks**
- The abstract instructions usually map 1-1 onto real ones
- In x86 there are extra constraints
  - *On combinations of operands*
  - *Some instructions (shift, multiply, divide) are special*
Code generation

- Code generator expression:
  \[ \text{asm } r : t = [e] \text{ in } a \]

- \( e \) is an IR expression (System F), \( a \) is an assembly expression

- to translate a program \( e \), start with
  \[ \text{asm } r : t = [e] \text{ in } %r \]

- Note: assembly types are different from IR, but not by much
Arithmetic

\[
\text{asm } r : t = [\nu] \text{ in } a \\
\rightarrow \text{ let } r : t = \%\nu \text{ in } a
\]

\[
\text{asm } r : \mathbb{Z} = [e_1 + e_2] \text{ in } a \\
\rightarrow \text{ asm } r_1 : \mathbb{Z} = [e_1] \text{ in } a \\
\text{asm } r_2 : \mathbb{Z} = [e_2] \text{ in } a \\
\text{let } r : \mathbb{Z} = \%r_1 + \%r_2 \text{ in } a
\]
**Tuple projection**

\[
\text{asm } r = \left[ \text{let} (x_1, \ldots, x_n) = e_1 \text{ in } e_2[x_1, \ldots, x_n] \right] \text{ in } a[r] \\
\rightarrow \text{asm } s = \left[ e_1 \right] \text{ in } \\
\text{let } x_1 = 0(\%s) \text{ in } \\
\ldots \\
\text{let } x_n = 4n(\%s) \text{ in } \\
\text{let } r = \left[ e_2[x_1, \ldots, x_n] \right] \text{ in } \\
a[r]
\]
**Tuple allocation**

- For type safety, we assume that malloc is an assembly primitive (like 1st generation TAL)

\[
asm \ r = \lfloor (e_1, \ldots, e_n) \rfloor \text{ in } a[r] \\
\rightarrow \quad asm \ r_1 = \lfloor e_1 \rfloor \text{ in} \\
\ldots \\
asm \ r_n = \lfloor e_n \rfloor \text{ in} \\
\text{let } r = \text{alloc}(\%r_1, \ldots, \%r_n) \text{ in } \# \text{ Cheat!} \\
a[r]
\]
Function call

\[
\begin{align*}
\text{asm } r &= \left[ e(e_1, \ldots, e_n) \right] \text{ in } a[r] \\
\rightarrow \text{asm } r_i &= \left[ e_i \right] \text{ in } \\
\ldots \\
\text{asm } r_c &= \left[ e \right] \text{ in } \\
\text{asm } r_f &= 0(\%r_c) \text{ in } \quad \# \text{ Function pointer} \\
\text{let } r &= (\star\%r_f)(\%r_c, \%r_1, \ldots, \%r_n) \text{ in } \\
a[r]
\end{align*}
\]
Step 2: register allocation

- After code generation, we have
  - an assembly program
  - using an unbounded number of variables/registers

![Register tile diagram]

- eax
- edx
- ecx
- ebx
- esi
- edi
- esp
- ebp

- 2-operand ALU
- L1 Cache (16k)
- L2 Cache (512k)
- Primary RAM (8MB-64GB)
Register allocation

- Use $\alpha$-renaming to use only register names for the variables
- There will be a lot of shadowing
- Formally, this is invisible!

\[
\begin{align*}
\text{let } f(r_1, r_2) &= \\
&= \text{let } r_3 = \%r_1 + \%r_2 \text{ in } \\
&\quad \text{let } r_4 = \%r_3 + 1 \text{ in } \\
&\quad \%r_4 \\
\rightarrow \quad \text{let } f(eax, ebx) &= \\
&= \text{let } eax = \%eax + \%ebx \text{ in } \\
&\quad \text{let } eax = \%eax + 1 \text{ in } \\
&\quad \%eax
\end{align*}
\]
Chaitin-style graph coloring

- Construct a graph with 1 node for each variable
- A variable is **live** from the point where it is defined, to the last point where it is used
- Two variables **interfere** iff they are both live at some program point
  - Add an edge between interfering variables
- Color the graph so adjacent vertices have different colors
  - A color stands for a register
Algorithm

FIR

build → simplify → coalesce → freeze → potential spill

select → actual spill → done
Spills

• Come back to reality!
• A real machine has a finite number of registers
  – (6)
• When too many variables are simultaneously live, some have to be “spilled”: stored in memory

\[
\text{let } r = e_1 \text{ in } e_2[r] \\
\rightarrow \text{let } r = e_1 \text{ in } \text{spill } s = r \text{ in } e_2[s]
\]
Spill optimization

• Each variable is:
  – defined once
  – then used 0-or-more times

• Split the range so that
  – the register is copied before each use
  – now only a portion of the live range may need to be spilled
Outline

"ML"  TAST  assembly

CPS  Closure  OPT  codegen  alloc

YOU ARE HERE

type inference
You made it!

- This is real x86 code
- The quality is good
  - *straightforward methods, about comparable to gcc -O1*
  - *Full employment theorem is still valid!*
- The formal part is *tiny!*
- The complete codebase is still comparable in size to traditional methods
  - *Register allocation, especially, is complicated*