Compiler Construction in Formal Logical Frameworks

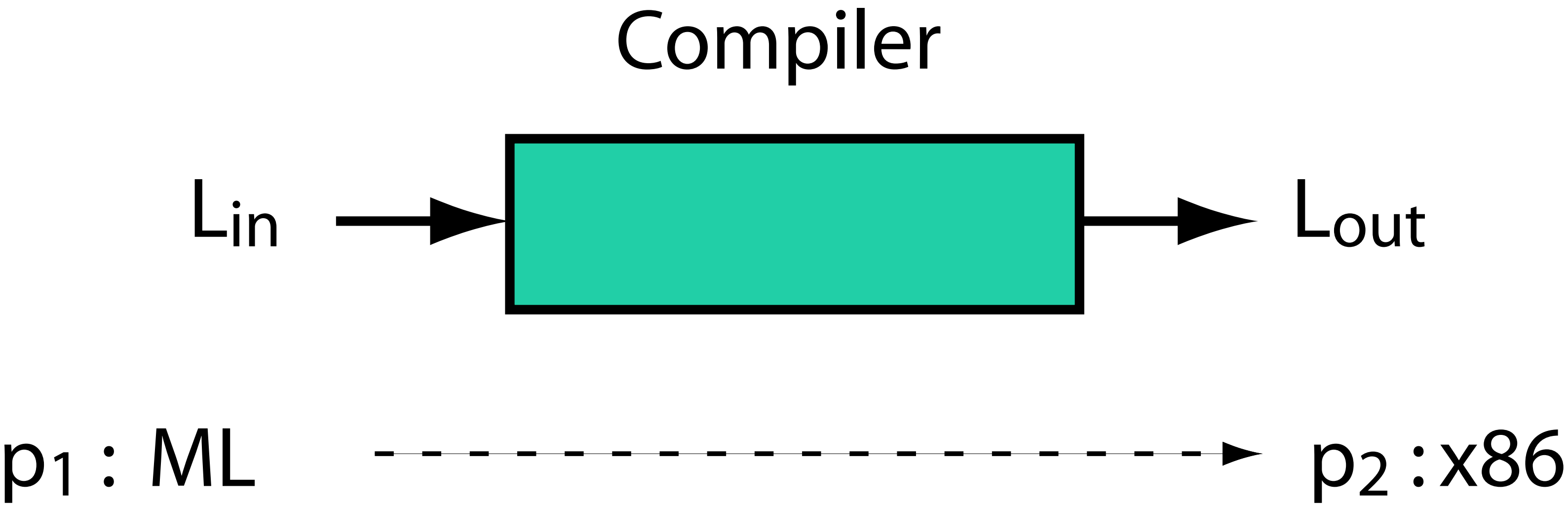
Jason Hickey
Caltech

Oregon Summer School on
Logic and Theorem Proving
in Programming Languages

Links

- MetaPRL: http://www.metaprl.org
- 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

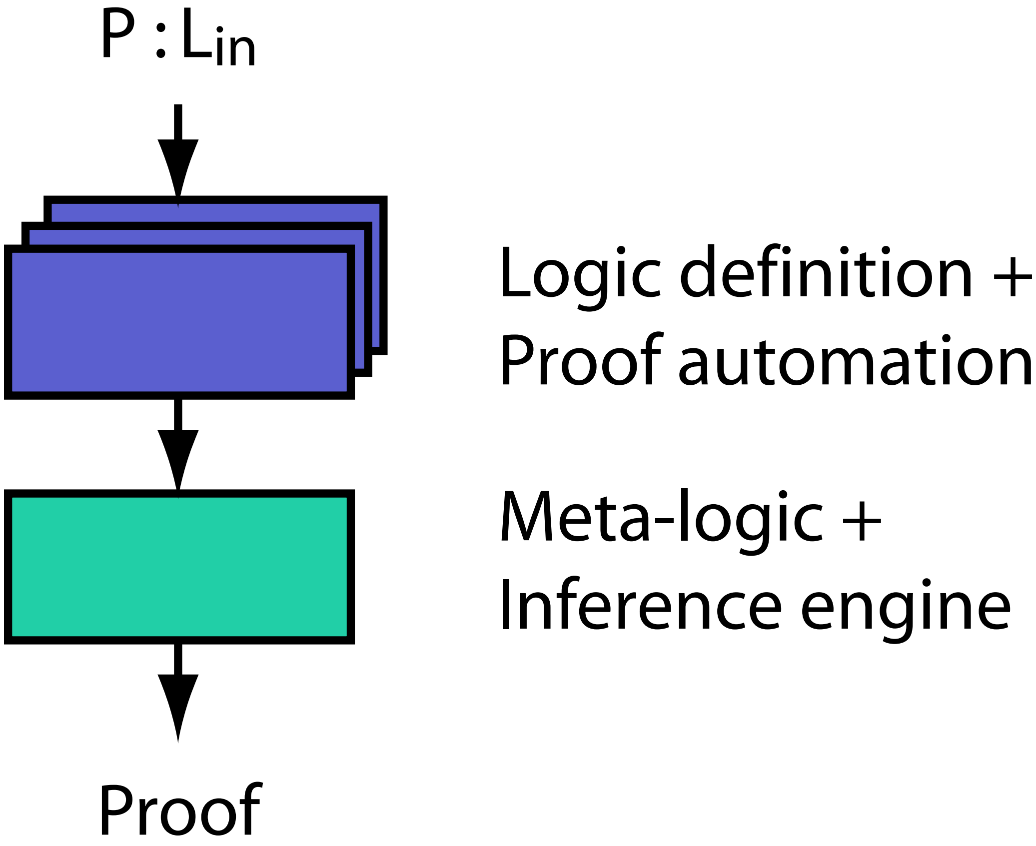
L_{in} → L_{out}

$p_1 : ML$ - - - → $p_2 : x86$

Requirement: $p_1 = p_2$

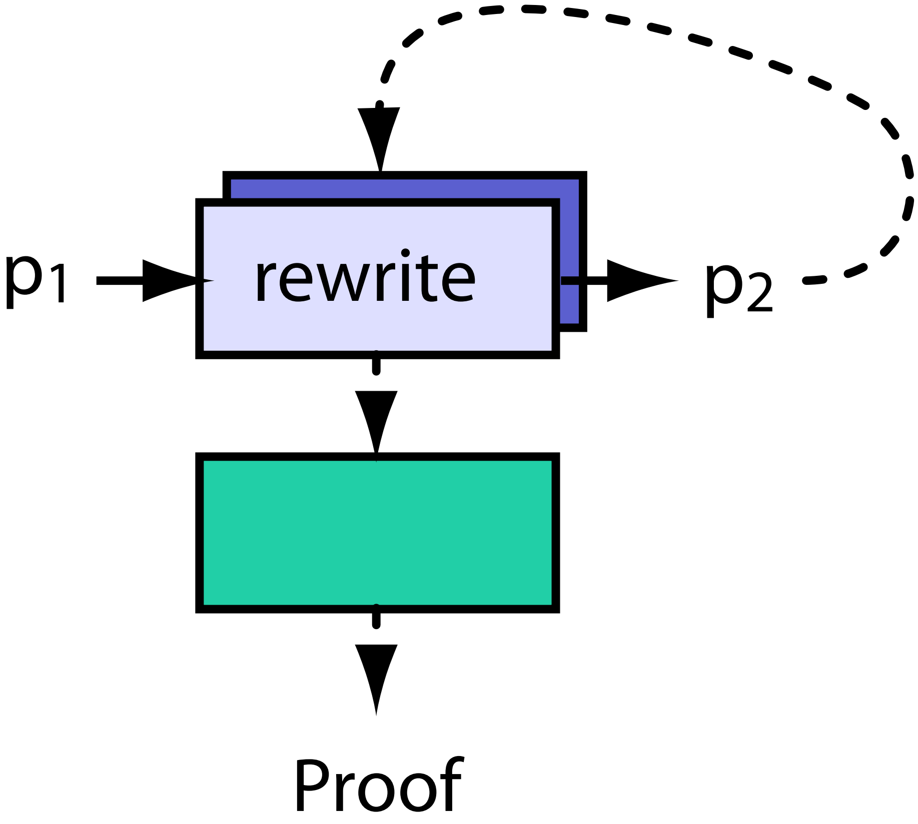
Logical Framework (highly simplified)

- Concepts:
  - *Judgments, inferences, proofs, program extraction, etc.*
- Techniques
  - *Refinement, term rewriting, tactics, search, etc.*
- LCF:
  - *Informal tactics in ML for proof automation*
  - *Proofs are <u>foundational</u>*



Plan

- Given p1, use term rewriting to find p2 s.t. p1=p2
- (for some p1, there exists p2. p1 = p2)
- NB: we will discuss certification, but we will focus primarily on program transformation

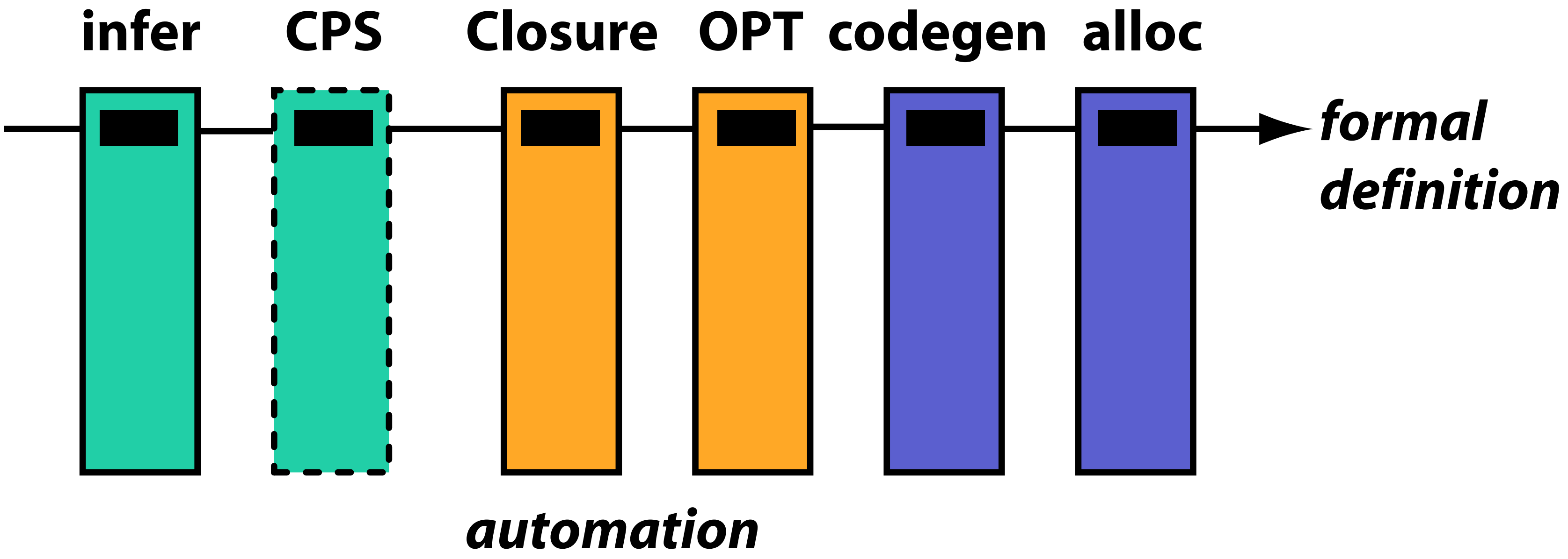


Why?

- 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

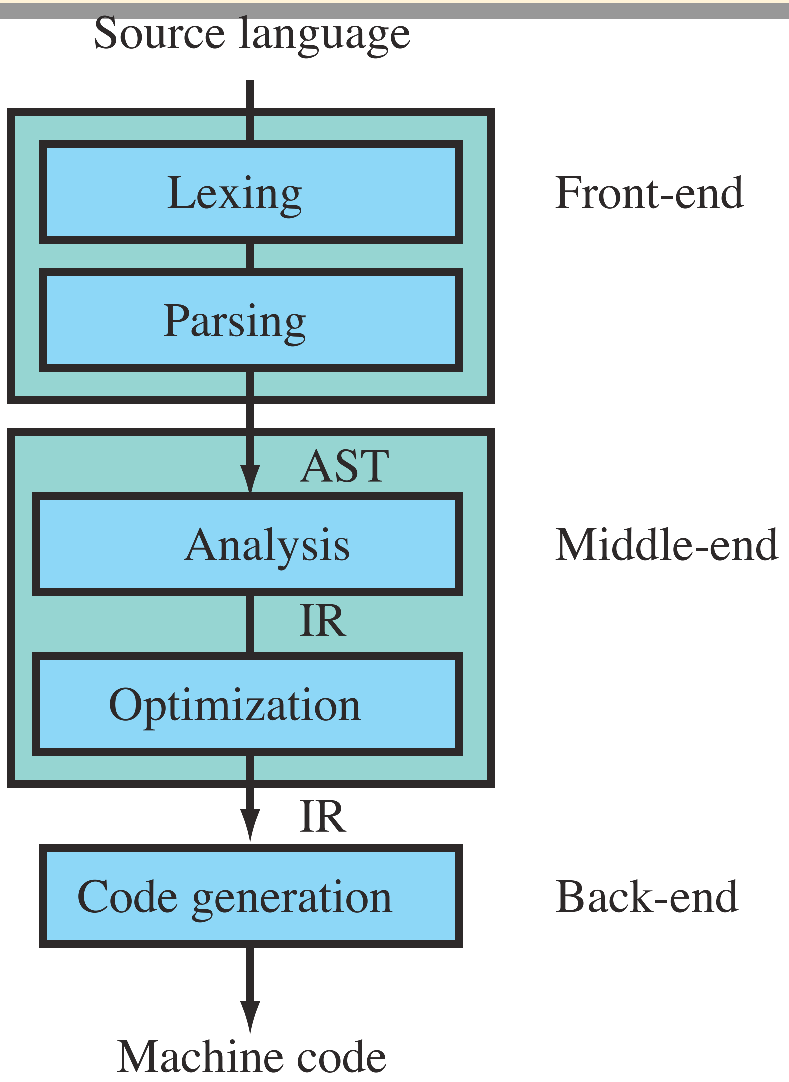
- Brian Aydemir’s undergraduate research project

- Aleksey Nogin, Nathan Gray, ...

Concerns

- Real compilers have many stages and many representations

- Compositionality is a fundamental concern



Source language

Lexing

Parsing

Front-end

AST

Analysis

Middle-end

IR

Optimization

IR

Code generation

Back-end

Machine code

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

MetaPRL: syntax

Explicit *term* syntax.

t	$::=$	$opname[p_1, \ldots, p_n]\{b_1; \cdots; b_m\}$	terms
	$\mid$	$x, y, z, \ldots$	variables
p	$::=$	$0, 1, 2, \ldots$	parameters (constants)
	$\mid$	"aaa", …	string constants
b	$::=$	$x_1, \ldots, x_n.t$	*bound* term

Syntax

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

Pretty form	Actual syntax
1	`number[1]{}`
$1 + 2$	`add{number[1]; number[2]}`
$\lambda x.x$	`lambda{x.x}`
$(\lambda x.x)\ 1$	`apply{lambda{x. x}; number[1]}`

Patterns and schemas

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

$$\begin{array}{rcll} t & ::= & \cdots & \text{terms} \\ & | & x[y_1; \cdots; y_n] & \end{array}$$

- 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 α-renaming convention)

Term	Pattern	Match
$y + y$	$x[y]$	$x[y] = y + y$
$y + z$	$x[y]$	**no match**
$1 + 2$	$x[y]$	$x[y] = 1 + 2$
$\lambda z.z + z$	$\lambda y.x[y]$	$x[y] = y + y$

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 β-equivalence:

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

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

- Rewrite application:

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

- where,

$$\begin{aligned} e_1[x] &= x + 1 \\ e_2[] &= 2 \\ e_1[e_2[]] &= 2 + 1 \end{aligned}$$

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

$$t_1 \longrightarrow t_2 \longrightarrow \cdots \longrightarrow t_n$$

- usually written like an inference rule

$$\frac{t_1 \quad t_2 \quad \cdots \quad t_{n-1}}{t_n}$$

- **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 $t_1 \longrightarrow \cdots \longrightarrow t_n \longrightarrow u$ is an axiom.
  - To prove $s_1 \longrightarrow \cdots \longrightarrow s_m \longrightarrow u$
  - You must prove $s_1 \longrightarrow \cdots \longrightarrow s_m \longrightarrow t_i$ 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

$$\frac{}{\Gamma; x{:}t; \Delta \vdash x : t}\ \text{var}$$

$$\frac{\Gamma \vdash e_1{:}s \to t \quad \Gamma \vdash e_2 : s}{\Gamma \vdash e_1\ e_2 : t}\ \text{app}$$

$$\frac{\Gamma, x{:}s \vdash e : t}{\Gamma \vdash (\lambda x{:}s.e) : s \to t}\ \text{abs}$$

- Context variables: Γ
- 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

$$s_1 \longrightarrow \cdots \longrightarrow s_n \longrightarrow (t_1 \longleftrightarrow t_2)$$

means t_1 and t_2 are equivalent in any context.

$$\begin{array}{l} s_1 \longrightarrow \cdots \longrightarrow s_n \longrightarrow \Gamma[t_1] \longrightarrow \Gamma[t_2] \\ s_1 \longrightarrow \cdots \longrightarrow s_n \longrightarrow \Gamma[t_2] \longrightarrow \Gamma[t_1] \end{array}$$

Dynamic semantics

- Rewriting axiom:

$$(\lambda x{:}t.e_1[x])\ e_2 \longleftrightarrow e_1[e_2]$$

- 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)

$$\frac{s_1 \quad \cdots \quad s_n}{t}\ \text{foo} \quad \Bigg| \quad \frac{\Gamma \vdash e_1 : S \to T \quad \Gamma \vdash e_2 : S}{\Gamma \vdash e_1\ e_2 : T}\ \text{app}$$

- Meta-rewrites

$$s \longleftrightarrow t \ \big|\ (\lambda x{:}S.e_1[x])\ e_2 \longleftrightarrow e_1[e_2]$$

Notes

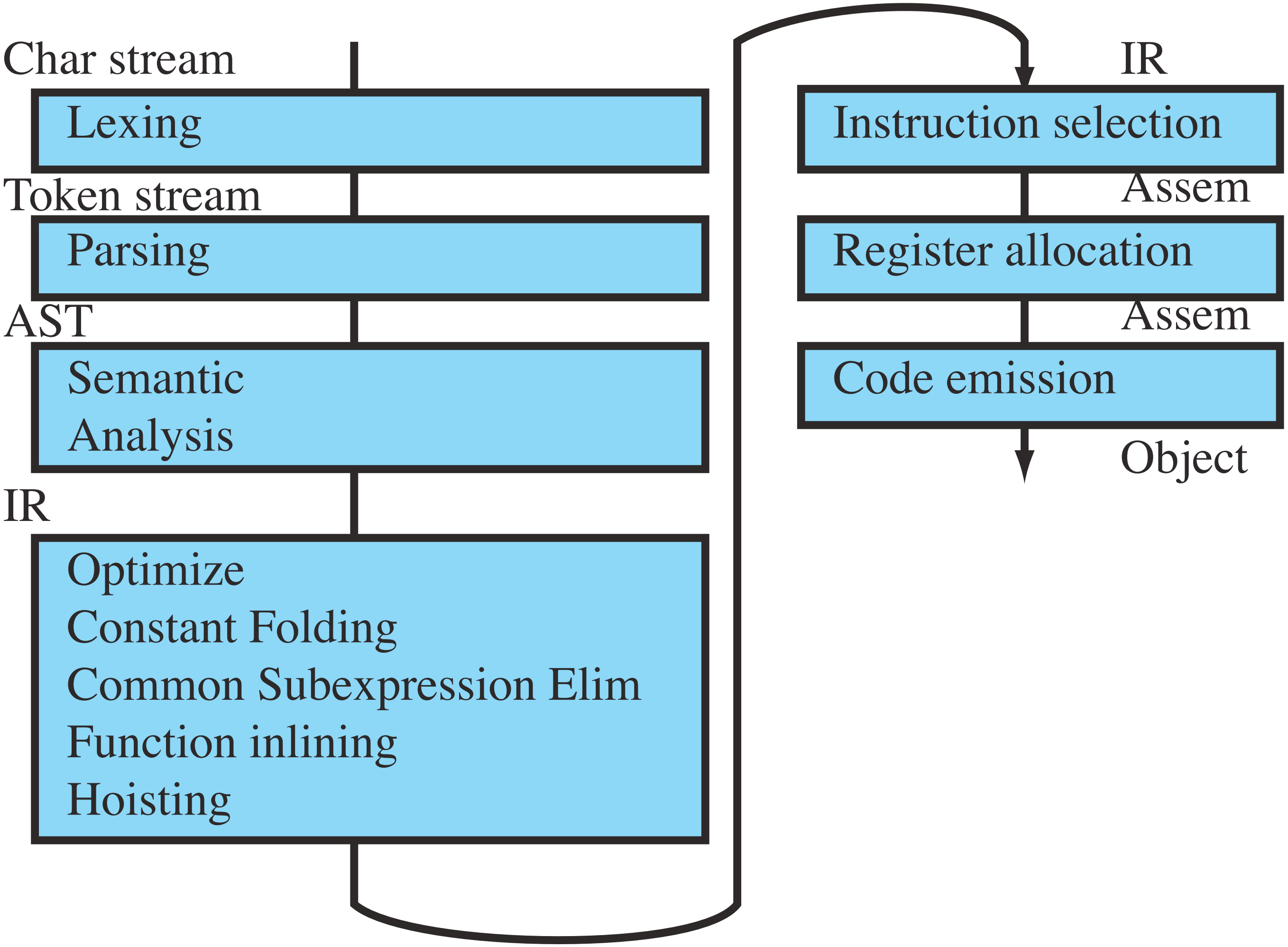
- Strictly speaking, context variables are *binders* and so-variables must specify them.

$$\frac{\Gamma \vdash e_1[\Gamma] : S[\Gamma] \to T[\Gamma] \quad \Gamma \vdash e_2[\Gamma] : S[\Gamma]}{\Gamma \vdash e_1[\Gamma]\; e_2[\Gamma] : T[\Gamma]}\ \text{app}$$

$$\Delta[(\lambda x{:}S[\Delta].e_1[x, \Delta])\; e_2[\Delta]] \longleftrightarrow \Delta[e_1[e_2[\Delta], \Delta]]$$

- 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

IR

Instruction selection

Assem

Register allocation

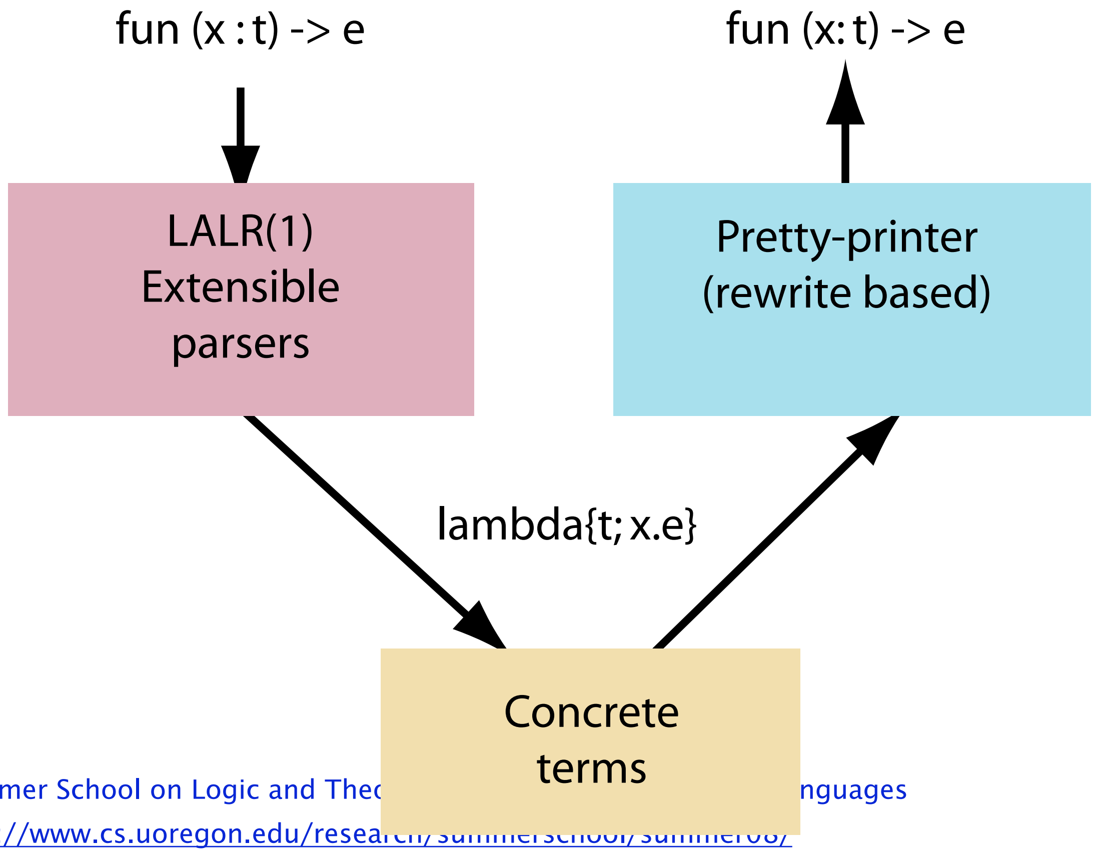
Assem

Code emission

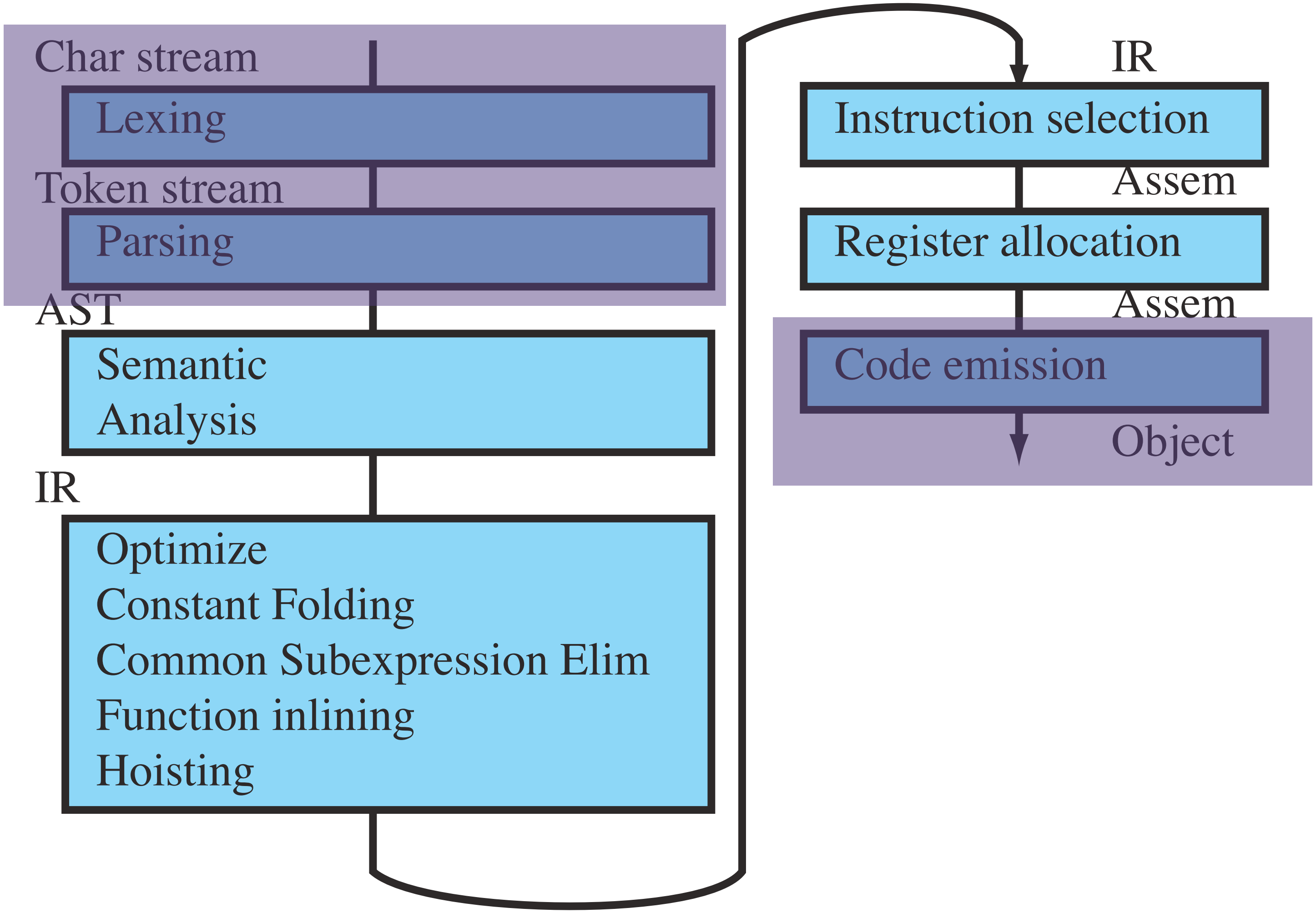
Object

Lexing, parsing, printing

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



Actual plan



Char stream

Lexing

Token stream

Parsing

AST

Semantic
Analysis

IR

Optimize
Constant Folding
Common Subexpression Elim
Function inlining
Hoisting

IR

Instruction selection

Assem

Register allocation

Assem

Code emission

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	$::=$		expressions
	$\mid$	$e(e_1, \ldots, e_n)$	application
	$\mid$	$\mathbf{fun}(x_1, \ldots, x_m) \to e$	function
	$\mid$	$\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2$	let
	$\mid$	$\mathbf{let\ rec}\ x_1 = e_1\ \mathbf{and} \cdots \mathbf{and}\ x_n = e_n\ \mathbf{in}\ e$	recursive definition

- Notes:
  - Arbitrary arity is achieved using *sequents*

$$\begin{array}{rcl} \mathbf{fun}(x_1, \ldots, x_n) \to e & \equiv & x_1{:}\_, \cdots, x_n{:}\_ \vdash_\lambda e \\ e(e_1, \ldots, e_n) & \equiv & e(\_{:}e_1, \ldots, \_{:}e_n \vdash_{args} \_) \end{array}$$

  - 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 *partial* argument
  - *The proof may fail because*
    - *our compiler is incorrectly automated*
    - *doesn’t terminate*
    - *the source program is “incorrect”*
  - *Translation validation: if a proof is found, it is correct*

- First step: how do we prove <<e>>?

Part II: types and type inference

- We'll use a *typed* intermediate language.
- Define a type erasure function *erase*,
- and a typed $\langle\langle e : t \rangle\rangle$ "compilable" judgment.

$$\frac{\vdash \langle\langle e : t \rangle\rangle}{\vdash \langle\langle erase(e) \rangle\rangle}\ \text{infer}$$

- automation: to compile an untyped program e,
  - find a typed program e' and a type t s.t. $e = erase(e')$ and $e' : t$.

Syntax: System F

t	$::=$	$\bot \mid \top$	base types
	$\mid$	$\mathbf{Fun}(t_1, \ldots, t_n) \rightarrow t$	function type
	$\mid$	$\forall(X_1, \ldots, X_n).t$	polymorphic type
e	$::=$	$e(e_1 : t_1, \ldots, e_n : t_n)$	application
	$\mid$	$\mathbf{fun}(x_1 : t_1, \ldots, x_n : t_n) \rightarrow e$	function
	$\mid$	$\mathbf{let}\ x : t = e_1\ \mathbf{in}\ e_2$	let
	$\mid$	$\mathbf{let\ rec}\ x_1 : t_1 = e_1$	recursive definition
		$\mathbf{and}\ \cdots$	
		$\mathbf{and}\ x_n : t_n = e_n$	
		$\mathbf{in}\ e$	
	$\mid$	$\mathbf{Lam}(X_1, \ldots, X_n) \rightarrow e$	type function
	$\mid$	$e[\vert t_1; \cdots; t_n \vert]$	type application

Type erasure

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

$$erase(\textbf{let}\, x : t = e_1 \textbf{ in } e_2) \longrightarrow \textbf{let}\, x = erase(e_1) \textbf{ in } erase(e_2)$$

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

$$erase(\textbf{fun}(x_1 : t_1, \ldots, x_n : t_n) \rightarrow e) \longrightarrow ???$$

Inductive definitions

- Introduce a temporary context $\Gamma \Vdash \cdots$, then specify the transformation by induction in 3 parts
- $\mathit{erase}(\textbf{fun}(\Delta) \to e) \longrightarrow \mathit{erase}(\Vdash \textbf{fun}(\Delta) \to e)$
- $\mathit{erase}(\Gamma \Vdash \textbf{fun}(x_i : t_i, \Delta) \to e) \longrightarrow$ $\mathit{erase}(\Gamma, x_i : \_ \Vdash \textbf{fun}(\Delta) \to e)$
- $\mathit{erase}(\Gamma \Vdash \textbf{fun}() \to e) \longrightarrow (\textbf{fun}(\Gamma) \to \mathit{erase}(e))$

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

$$\frac{\Gamma \vdash e_1 : t \quad \Gamma, x{:}t \vdash e_2[x] : s}{\Gamma \vdash (\mathbf{let}\, x : t = e_1\ \mathbf{in}\ e_2[x]) : s}\ \text{let}$$

Type checking unbounded arity

$$\frac{\Gamma \vdash e : t}{\Gamma \vdash (\mathbf{fun}() \to e) : (\mathbf{Fun}() \to t)}\ \text{fun0}$$

$$\frac{\Gamma, x{:}s \vdash (\mathbf{Fun}(\Delta_1) \to e) : (\mathbf{Fun}(\Delta_2) \to t)}{\Gamma \vdash (\mathbf{fun}(x : s, \Delta_1) \to e) : (\mathbf{Fun}(s, \Delta_2) \to t)}\ \text{fun1}$$

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

$$\frac{\Gamma, X \vdash (\mathbf{Lam}(\Delta_1) \to e) : (\forall(\Delta_2) \to t)}{\Gamma \vdash (\mathbf{Lam}(X, \Delta_1) \to e) : (\forall(X, \Delta_2) \to t)}\ \mathrm{Lam1}$$

Type inference

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

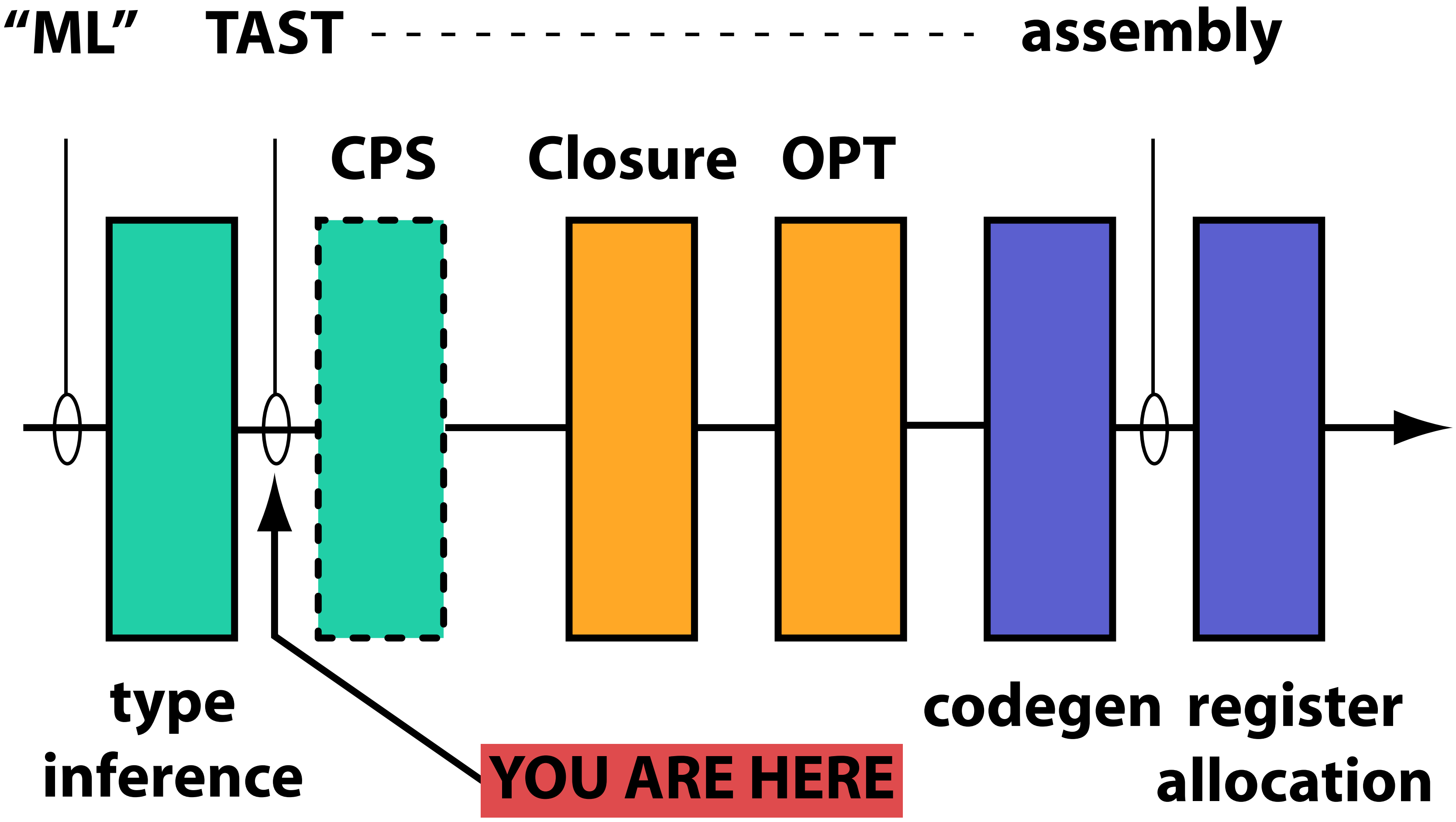
$$\frac{\vdash \langle\langle e : t \rangle\rangle}{\vdash \langle\langle erase(e) \rangle\rangle} \text{ infer}$$

- 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. $\mathit{erase}(e') = e$ 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



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)

$$
\begin{aligned}
&[\![\mathbf{let}\, x = N \;\mathbf{in}\; M]\!] = \\
&\qquad\qquad \overline{\lambda} c.\overline{@}[\![N]\!](\overline{\lambda} n.\, \underline{\mathbf{let}}\, x' = n \;\underline{\mathbf{in}}\; \overline{@}[\![M\,[x \leftarrow x']]\!]c)
\end{aligned}
$$

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

$$
\begin{aligned}
&\mathsf{CPS}\{\mathbf{let}\, v_1 : t_1 = e_1 \;\mathbf{in}\; e_2[v_1]; t_2; v_2.c[v_2]\} \\
&\quad \longleftarrow [\mathsf{cps\_let}] \longrightarrow \\
&\mathsf{CPS}\{e_1; t_1; v_3.\mathbf{let}\, v_1 : \mathsf{TyCPS}\{t_1\} = v_3 \;\mathbf{in} \\
&\qquad\qquad\qquad \mathsf{CPS}\{e_2[v_1]; t_2; v_2.c[v_2]\}\,\}
\end{aligned}
$$

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

$$\cdots (\mathbf{fun}(x : t) \to x + y) \cdots$$

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

$$\cdots (\mathbf{let}\, y : \mathbb{Z} = y \;\mathbf{in}\; \mathbf{fun}(x : \mathbb{Z}) \to x + y) \cdots$$

parts 2 and 3

2. Add an extra parameter, and apply it

$$\cdots(\mathbf{let}\, y : \mathbb{Z} = y\ \mathbf{in}\ (\mathbf{fun}(y : \mathbb{Z}, x : \mathbb{Z}) \to x+y)(y))\cdots$$

3. Hoist

$$\begin{array}{l} \mathbf{let}\, f = \mathbf{fun}(y : \mathbb{Z}, x : \mathbb{Z}) \to x + y\ \mathbf{in} \\ \cdots(\mathbf{let}\, y : \mathbb{Z} = y\ \mathbf{in}\ f(y))\cdots \end{array}$$

Formal definition (parts 2, 3)

2. Purely local definition

$$\begin{aligned}&\textbf{let}\, x{:}t = e_1 \textbf{ in } \textbf{fun}(\Delta) \to e_2[x]\\ \longleftrightarrow\ &\textbf{let}\, x{:}t = e_1 \textbf{ in } (\textbf{fun}(x{:}t, \Delta) \to e[x])(x : t)\end{aligned}$$

3. Need a single context

$$\textbf{let}\, f = e[\,] \textbf{ in } \Gamma[f] \longleftrightarrow \Gamma[e[\,]]$$

- $\Gamma[e[\,]]$ 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!**

$$e[x] \longleftrightarrow \textbf{let}\, x\!:\!t = x \;\textbf{in}\; e[x]$$

- 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.

$$\begin{array}{l}\textbf{let}\, x : t = e_1 \textbf{ in } \Gamma[e[x]] \\ \longleftrightarrow \\ \textbf{let}\, x : t = x_1 \textbf{ in} \\ \Gamma[\textbf{let}\, x : t = x \textbf{ in } e[x]]\end{array}$$

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

$$sweep(\Sigma \Vdash e)$$

- where Σ is a set of membership terms

$$\Sigma ::= x_1 \in t_1, \ldots, x_n \in t_n$$

Definition

The general form of $sweep(\Sigma \Vdash e)$ is defined by structural induction

$$
\begin{aligned}
& sweep(\Sigma \Vdash \mathbf{let}\, x{:}t = e_1 \,\mathbf{in}\, e_2) \\
& \longleftrightarrow \mathbf{let}\, x{:}t = sweep(\Sigma \Vdash e_1) \,\mathbf{in}\, sweep(\Sigma, x{:}t \Vdash e_2)
\end{aligned}
$$

$$
\begin{aligned}
& sweep(\Sigma \Vdash \mathbf{fun}(\Delta) \to e) \\
& \longleftrightarrow \mathbf{fun}(\Delta) \to sweep(\Sigma, \Delta \Vdash e)
\end{aligned}
$$

$$
\begin{aligned}
& sweep(\Sigma \Vdash e(e_1, \ldots, e_n)) \\
& \longleftrightarrow sweep(\Sigma \Vdash e)(sweep(\Sigma \Vdash e1), \ldots, sweep(\Sigma \Vdash e_n))
\end{aligned}
$$

$$sweep(\Sigma \Vdash x) \longleftrightarrow x$$

Sweep let droppings

- *Generic* rule for adding a let-definition

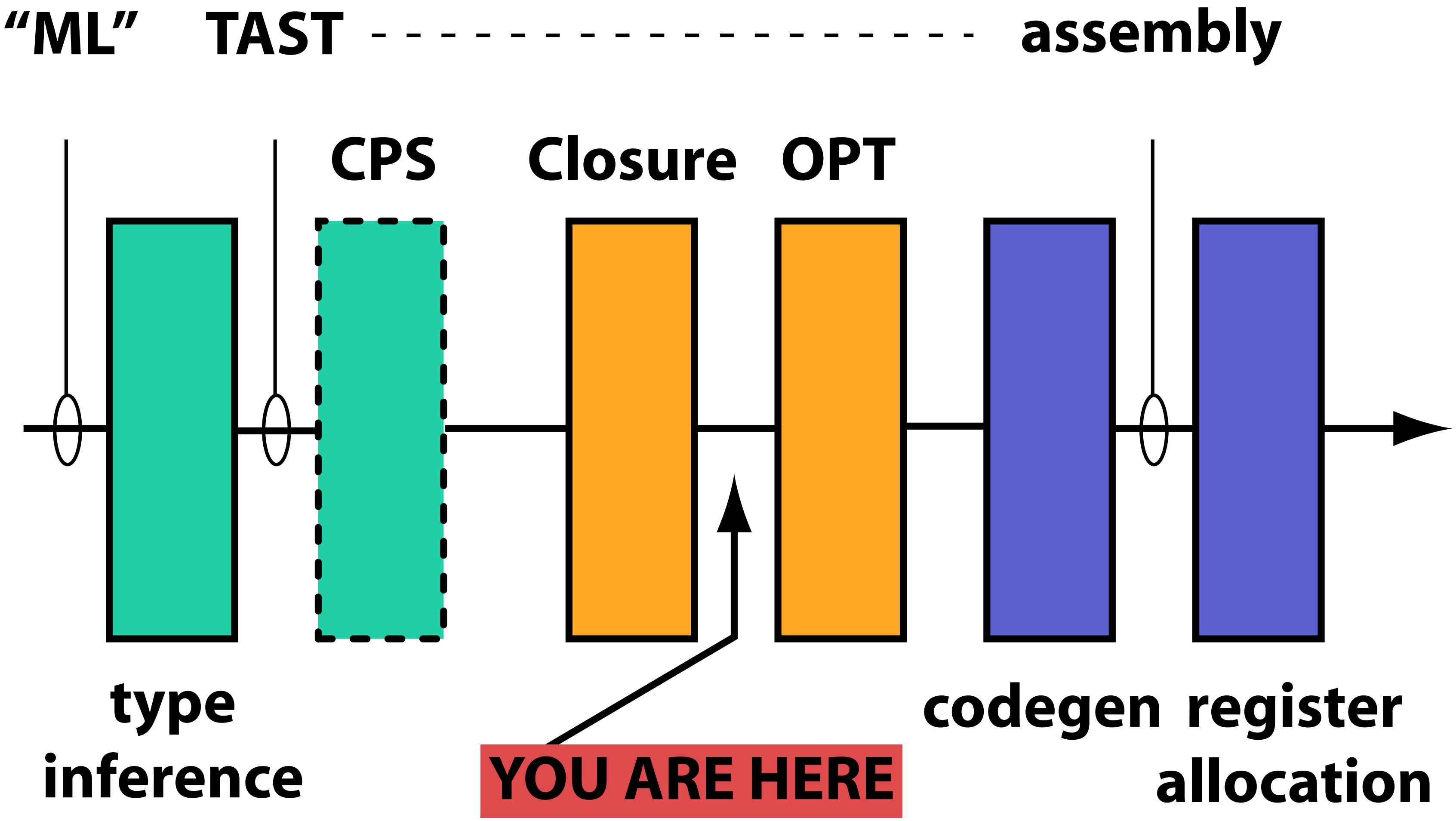
$$
\begin{aligned}
&sweep(\Sigma_1, x \in t, \Sigma_2 \Vdash e[x]) \\
&\longleftrightarrow sweep(\Sigma_1, x \in t, \Sigma_2 \Vdash \mathbf{let}\, x : t = x \,\mathbf{in}\, e[x])
\end{aligned}
$$

- 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:

$$\textbf{let}\, x\!:\!t = e_1 \textbf{ in } e_2 \longrightarrow e_2$$

(note x is not free in e_2)

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

Common subexpression elimination

- Inverse beta-reduction (if language is pure)

$$\mathbf{let}\, x : t = e_1\ \mathbf{in}\ e_2[x] \longleftarrow e_1[e_2]$$

- Apply in reverse (right-to-left)

$$a * b + f(a * b)$$
$$\mathbf{let}\, x : \mathbb{Z} = a * b\ \mathbf{in} \cdots x + f(x)$$

Inlining

- (beta-reduction)

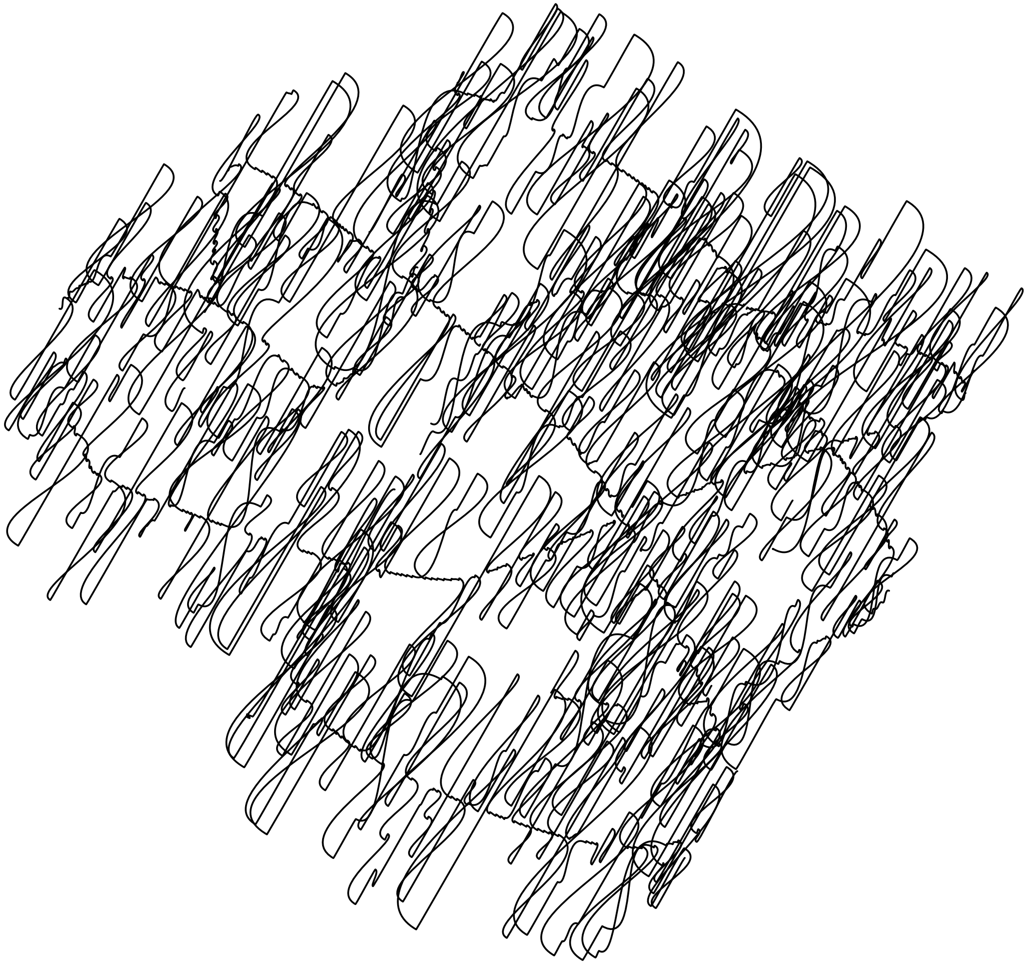
$$\textbf{let}\, x : t = e_1 \textbf{ in } e_2[x] \quad \longrightarrow \quad e_2[e_1]$$

$$(\textbf{fun}(x : t, \Delta_1) \to e[x])(e_1, \Delta_2) \quad \longrightarrow \quad (\textbf{fun}(\Delta_1) \to e[e_1])(\Delta_2)$$

- Example:

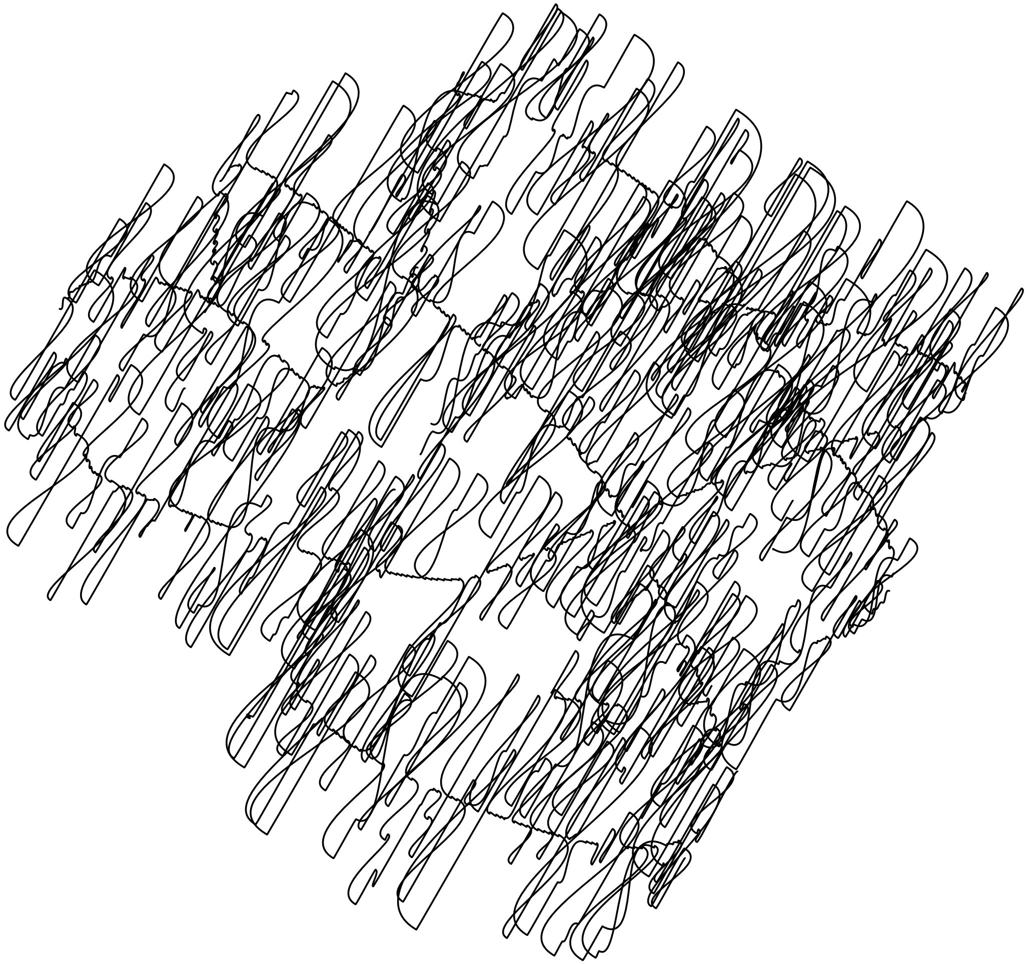
$$\begin{aligned}
&\textbf{let}\, f = \textbf{fun}(x : \mathbb{Z}) \to x + x \textbf{ in } f(1)\\
&\longrightarrow (\textbf{fun}(x : \mathbb{Z}) \to x + x)(1)\\
&\longrightarrow (\textbf{fun}() \to 1 + 1)()\\
&\longrightarrow 1 + 1
\end{aligned}$$

Partial Redundancy Elimination

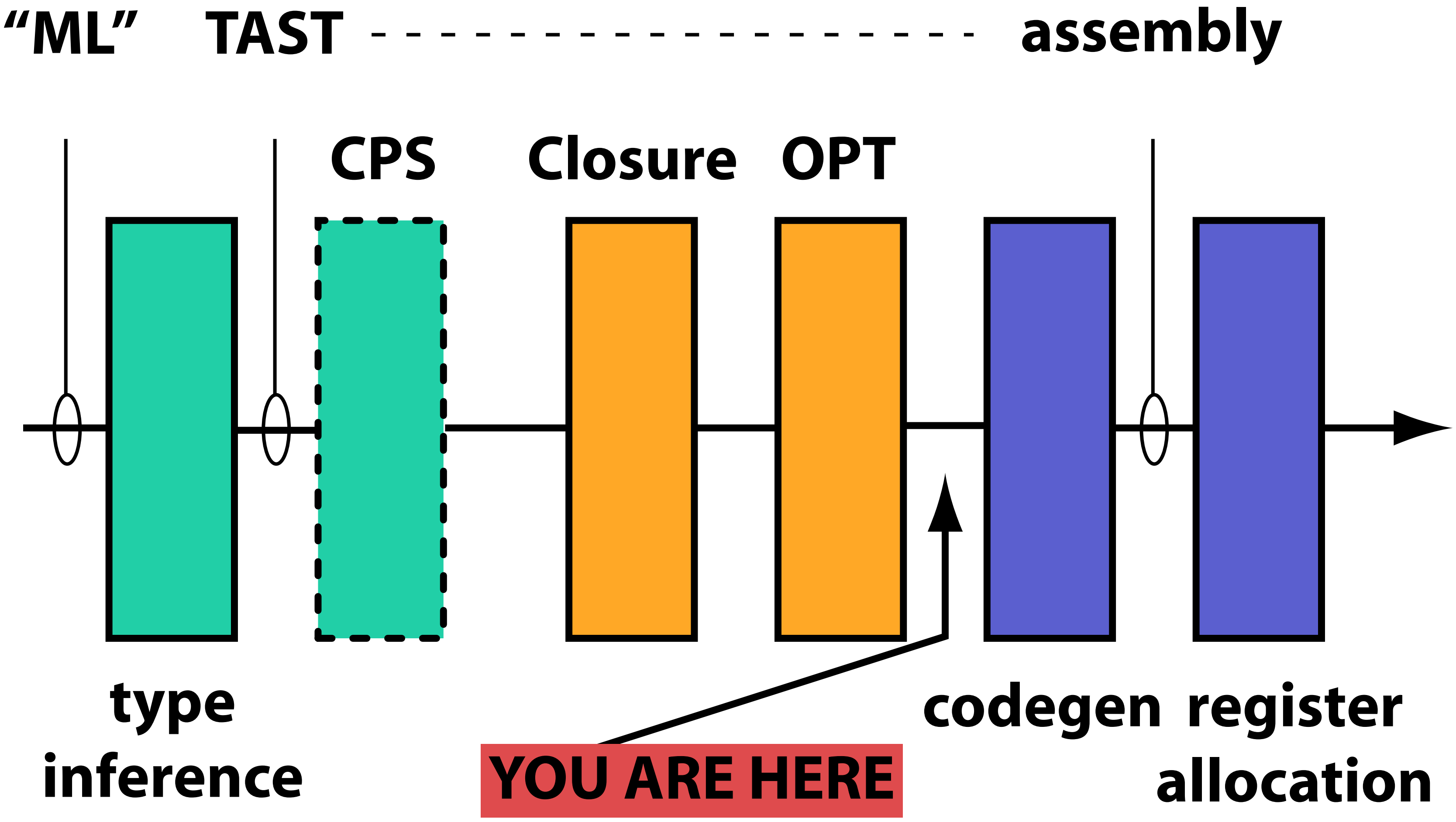


Partial Redundancy Elimination

- So there, Sorin!



Outline


"ML"
TAST
assembly
CPS
Closure
OPT
type inference
YOU ARE HERE
codegen
register allocation

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)*

$$\begin{array}{l}\textbf{let rec } f_1 : t_1 = e_1\\ \textbf{and } \cdots\\ \textbf{and } f_n : t_n = e_n\\ \textbf{in } e\end{array}$$

Step 1: add a let-definition (simultaneous)

- Collect free variables

$$\begin{aligned}&\textbf{let}\ \Delta_1 = \Delta_2\ \textbf{in}\\ &\textbf{let rec}\ f_1 : t_1 = e_1\\ &\textbf{and}\ \cdots\\ &\textbf{and}\ f_n : t_n = e_n\\ &\textbf{in}\ e\end{aligned}$$

- $\Delta_1 = (x_1{:}t_1, \ldots, x_m{:}t_m)$
- $\Delta_2 = (x_1, \ldots, x_m) = FV(e_1) \cup \cdots \cup FV(e_n)$

Step 2: Add extra function parameters

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

$$
\begin{array}{l}
\textbf{let}\ \Delta_1 = \Delta_2\ \textbf{in} \\
\textbf{let rec}\ f'_1 : \textbf{Fun}(\Delta_1) \to t_1 = \textbf{fun}(\Delta_1) \to e_1 \\
\textbf{and}\ \cdots \\
\textbf{and}\ f'_n : \textbf{Fun}(\Delta_1) \to t_n = \textbf{fun}(\Delta_1) \to e_n \\
\textbf{and}\ f_1 : t_1 = f'_1(\Delta_2) \\
\textbf{and}\ \cdots \\
\textbf{and}\ f_n : t_n = f'_n(\Delta_2) \\
\textbf{in}\ e
\end{array}
$$

Notes:

- This is actually done 1 function at a time
  - Close f_1 in **let rec** $f_1, \ldots f_n$ **in** $\cdots$
  - Then rotate to **let rec** $f_2, \ldots, f_n, f'_1, f_1$ **in** $\cdots$
- In a real compiler, only 1 closure is needed:
  - $c = (f'_1, \ldots, f'_n, x_1, \ldots, x_m)$
  - $f_i(e, \ldots, e) = apply_i(c, e, \ldots, e)$
  - Easy to do (but the language needs to be extended)

Pretty important optimization

- Inline closures when possible

$$
\begin{aligned}
&\textbf{let}\, c = f(e_1, \ldots, e_m)_c \textbf{ in } \Delta[c(e_{m+1}, \ldots, e_n)] \\
&\longrightarrow \textbf{let}\, c = f(e_1, \ldots, e_m)_c \textbf{ in} \\
&\qquad \Delta[f(e_1, \ldots, e_m, e_{m+1}, \ldots, e_n)_d]
\end{aligned}
$$

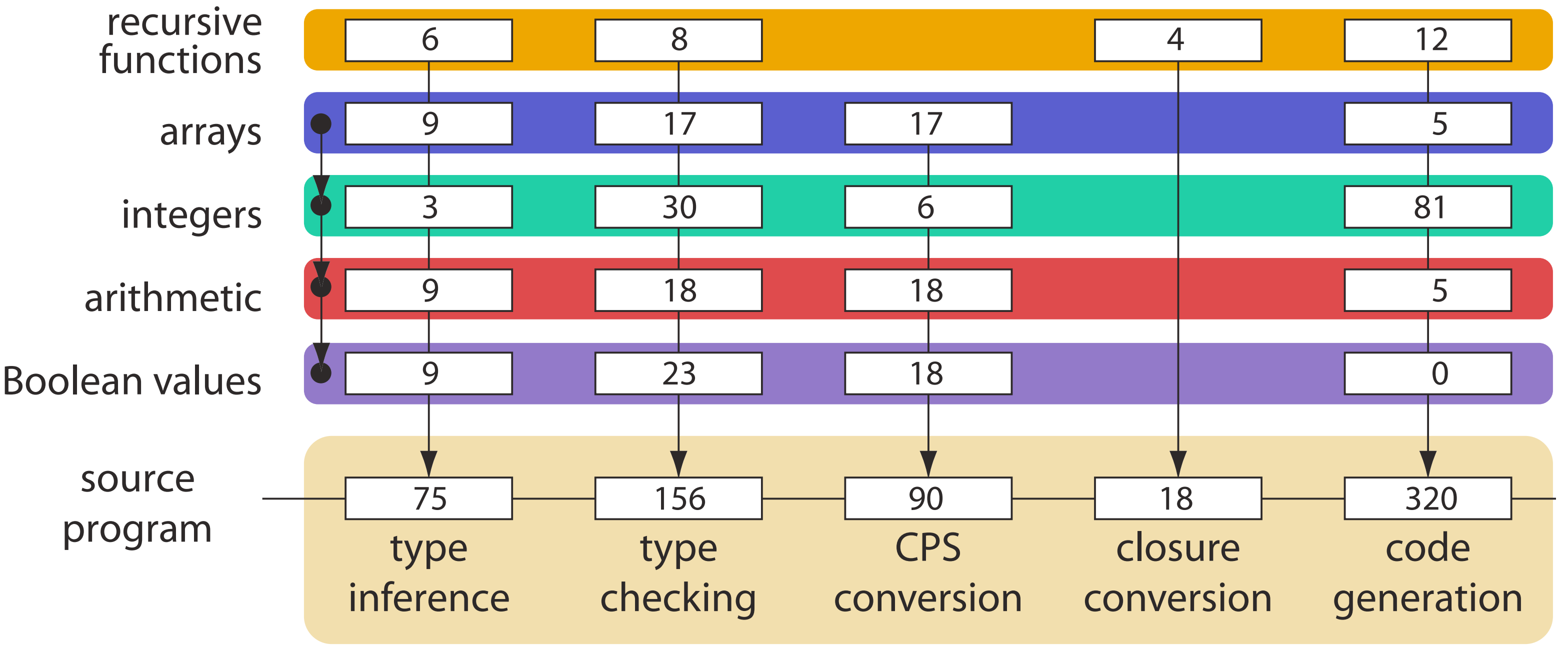
Extensibility, compositionality

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



	type inference	type checking	CPS conversion	closure conversion	code generation
recursive functions	6	8		4	12
arrays	9	17	17		5
integers	3	30	6		81
arithmetic	9	18	18		5
Boolean values	9	23	18		0
source program	75	156	90	18	320

core: polymorphic lambda calculus

Useful example: Tuples

- New syntax
  - *(Note: MetaPRL grammars are extensible)*
- Untyped language

$$
\begin{array}{lcll}
e & ::= & \cdots & \text{expressions} \\
 & | & (e_1, \ldots, e_n) & \text{tuple} \\
 & | & \mathbf{let}\,(x_1, \ldots, x_n) = e_1 \;\mathbf{in}\; e_2 & \text{projection}
\end{array}
$$

Tuples: typed language

t	$::=$	$\cdots$	types
	$\mid$	$t_1 * \cdots * t_n$	product type
e	$::=$	$\cdots$	expressions
	$\mid$	$(e_1 : t_1, \ldots, e_n : t_n)$	tuple
	$\mid$	$\mathbf{let}(x_1{:}t_1, \ldots, x_n{:}t_n) = e_1 \ \mathbf{in}\ e_2$	projection

Tuple: type erasure

- Erasure definition

$$\begin{array}{rcl} \mathit{erase}(e_1 : t_1, \ldots, e_n : t_n) & \longrightarrow & (\mathit{erase}(e_1), \ldots, \mathit{erase}(e_n)) \\ \mathit{erase}(\mathbf{let}(x_1 : t_1, \ldots, x_n : t_n) = e_1 \ \mathbf{in}\ e_2) & \longrightarrow & \mathbf{let}(x_1, \ldots, x_n) = \mathit{erase}(e_1)\ \mathbf{in}\ \mathit{erase}(e_2) \end{array}$$

- Automation is still automatic (just include these rewrites).

Tuple: type checking

$$\frac{}{\Gamma \vdash () : ()}\ \text{tuple}_0$$

$$\frac{\Gamma \vdash e : t \quad \Gamma \vdash (\Delta_1) : \Delta_2}{\Gamma \vdash (e : t, \Delta_1) : t * \Delta_2}\ \text{tuple}_1$$

$$\frac{\Gamma \vdash e_1 : (\Delta) \quad \Gamma, \Delta \vdash e_2 : t}{\Gamma \vdash (\mathbf{let}(\Delta) = e_1\ \mathbf{in}\ e_2) : t}\ \text{proj}$$

Tuple: sweep (for closure conversion)

$$sweep(\Sigma \Vdash (e_1 : t_1, \ldots, e_n : t_n)$$
$$\longrightarrow (sweep(\Sigma \Vdash e_1) : t_1, \ldots, sweep(\Sigma \Vdash e_n) : t_n)$$

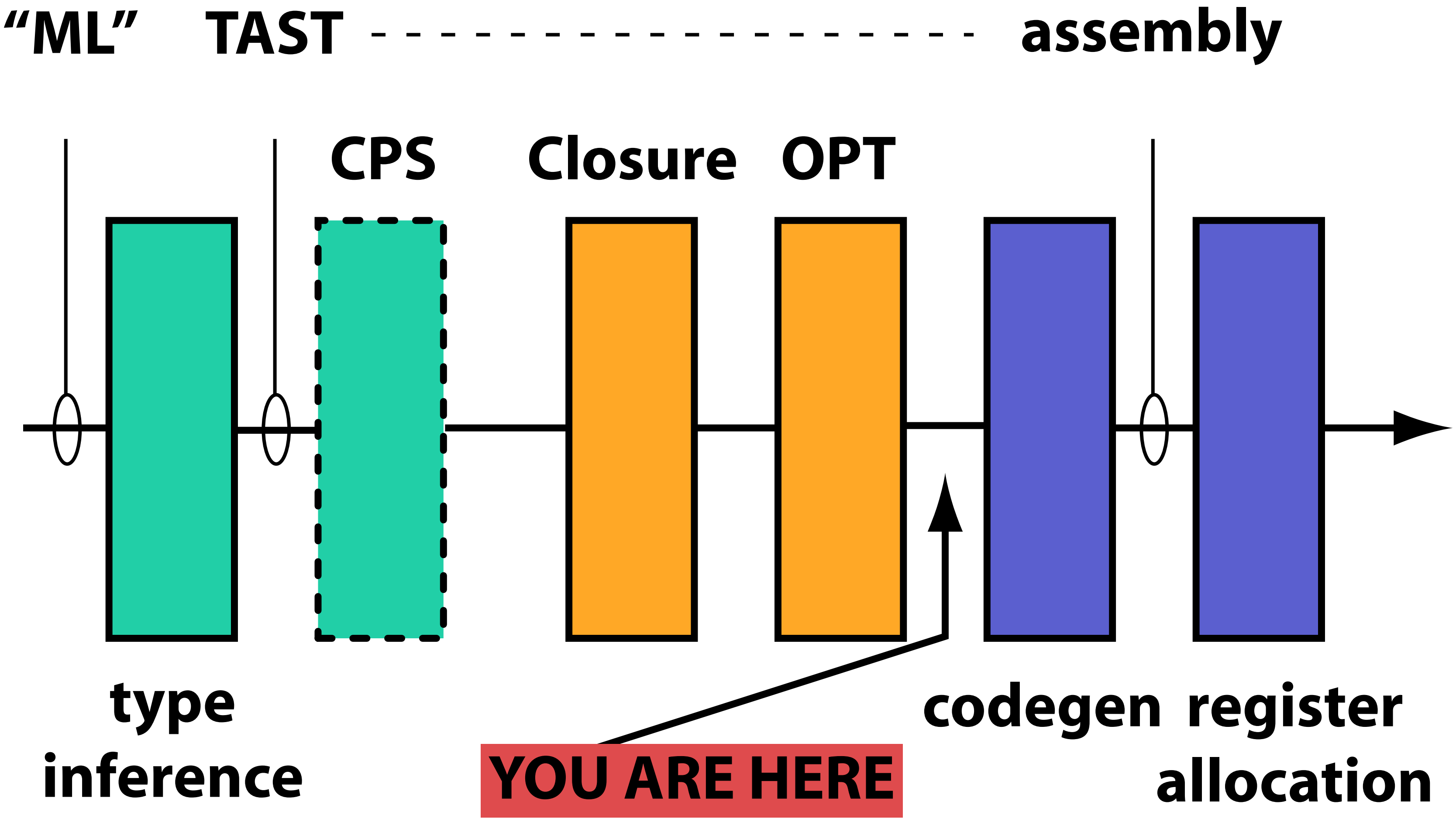
$$sweep(\Sigma \Vdash \mathbf{let}(\Delta) = e_1 \ \mathbf{in}\ e_2)$$
$$\longrightarrow \mathbf{let}(\Delta) = sweep(\Sigma \Vdash e_1) \ \mathbf{in}\ sweep(\Sigma \Vdash e_1)$$

Revisiting closure conversion

- Represent the environment as a tuple

$$
\begin{aligned}
&\mathbf{let}(\Delta_1) = (\Delta_2)\ \mathbf{in}\ \mathbf{fun}(\Delta_3) \to e[\Delta_1, \Delta_3] \\
\longleftrightarrow\quad &\mathbf{let}(\Delta_1) = (\Delta_2)\ \mathbf{in} \\
&\quad (\mathbf{fun}(x : \cdot, \Delta_3) \to \\
&\qquad \mathbf{let}(\Delta_1) = x\ \mathbf{in}\ e[\Delta_1, \Delta_3])((\Delta_2))
\end{aligned}
$$

Outline


"ML"
TAST
assembly
CPS
Closure
OPT
type inference
YOU ARE HERE
codegen
register allocation

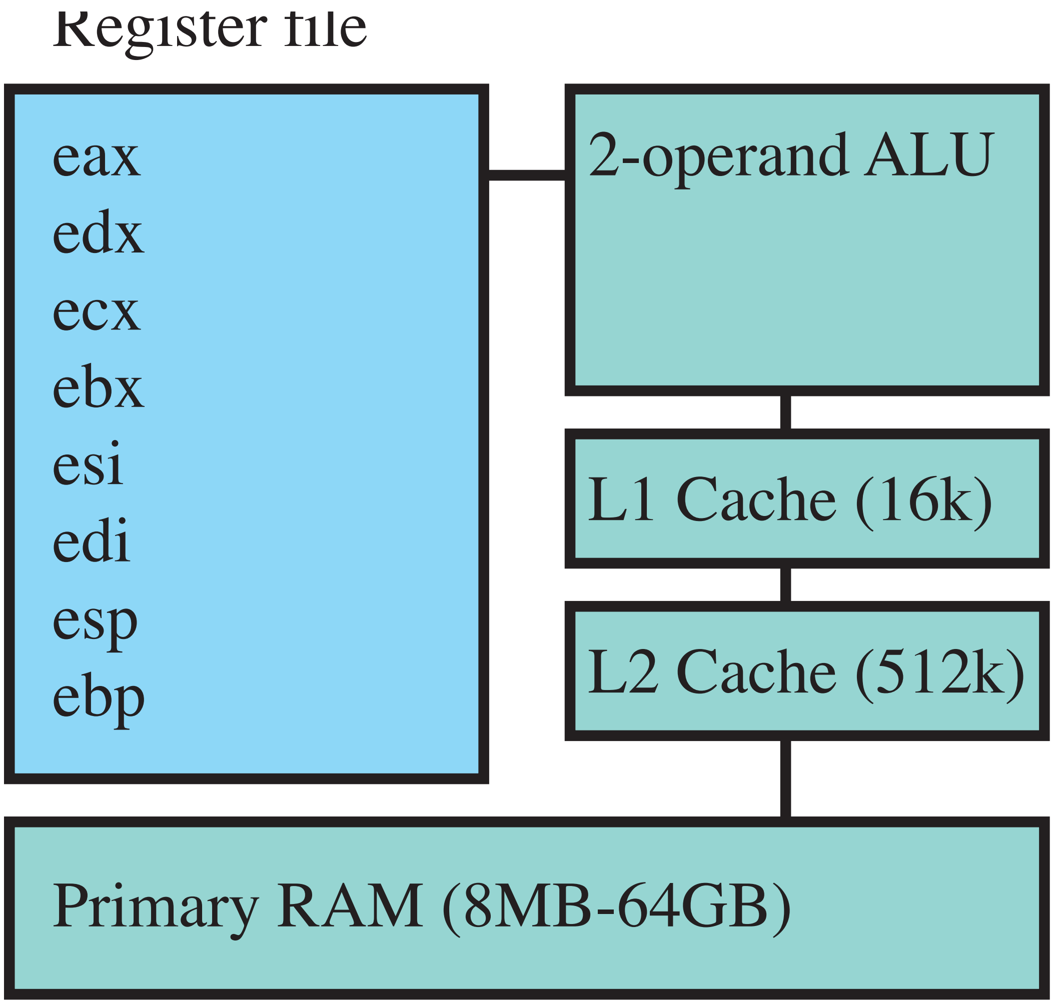
Code generation

- 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-operand 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
  - *ADD op1,op2*
    - *means op1 += op2*
- Some instructions are strange
  - *MUL op1*
    - *means (edx,eax) *= op1*
  - *SHL op1,op2*
    - *means op1 <<= op2*
    - *but 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

$$\mathit{opcode} \quad \mathit{operand}_1, \mathit{operand}_2$$

- Operand

$\mathit{operand}$	$::=$	i	address
	$\mid$	$\$i$	integer constant
	$\mid$	$\%r$	register
	$\mid$	$(\%r)$	indirect – `*r`
	$\mid$	$i(\%r)$	offset – `*(r + i)`
	$\mid$	$i_1(\%r_1, \%r_2, i_2)$	`*(r1 + r2*i2 + i1)`

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 <u>needs</u> 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

$$
\begin{array}{rcll}
e & ::= & \textbf{let}\, r : t \,=\, op \;\textbf{in}\; e & \text{Load} \\
  & | & op \leftarrow \%r ; e & \text{Store} \\
  & | & \textbf{let}\, r : t \,=\, op_1 + op_2 \;\textbf{in}\; e & \text{arithmetic} \\
  & | & \textbf{let}\, r : t \,=\, f(r_1, \ldots, r_n) \;\textbf{in}\; e & \text{function call} \\
  & | & \mathsf{jmp}\, f(r_1, \ldots, r_2) & \text{unconditional branch} \\
  & | & \mathsf{cmp}\, op_1, op_2 ; e & \text{compare} \\
  & | & \textbf{if}\; cc \;\textbf{then}\; e_1 \;\textbf{else}\; e_2 & \\
  & | & \mathsf{ret}\, op & \\
  &   & & \\
p & ::= & \textbf{let rec}\; f_1(r, \ldots, r) = e_1 & \\
  &   & \textbf{and}\; f_2(r, \ldots, r) = e_2 & \\
  &   & \ldots & \\
  &   & \textbf{and}\; f_n(r, \ldots, r) = e_n &
\end{array}
$$

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:

$$\textbf{asm}\, r : t = [\![e]\!]\ \textbf{in}\ a$$

- e is an IR expression (System F), a is an assembly expression
- to translate a program e, start with $\textbf{asm}\, r : t = [\![e]\!]\ \textbf{in}\ \%r$
- Note: assembly types are different from IR, but not by much

Arithmetic

$$\textbf{asm}\ r : t = [\![v]\!]\ \textbf{in}\ a$$
$$\longrightarrow \textbf{let}\ r : t = \%v\ \textbf{in}\ a$$

$$\textbf{asm}\ r : \mathbb{Z} = [\![e_1 + e_2]\!]\ \textbf{in}\ a$$
$$\begin{aligned}\longrightarrow\ & \textbf{asm}\ r_1 : \mathbb{Z} = [\![e_1]\!]\ \textbf{in} \\ & \textbf{asm}\ r_2 : \mathbb{Z} = [\![e_2]\!]\ \textbf{in} \\ & \textbf{let}\ r : \mathbb{Z} = \%r_1 + \%r_2\ \textbf{in} \\ & a\end{aligned}$$

Tuple projection

$$
\begin{array}{ll}
\multicolumn{2}{l}{\textbf{asm}\, r = [\![\textbf{let}(x_1,\ldots,x_n) = e_1 \textbf{ in } e_2[x_1,\ldots,x_n]]\!] \textbf{ in } a[r]} \\
\longrightarrow & \textbf{asm}\, s = [\![e_1]\!] \textbf{ in} \\
 & \textbf{let}\, x_1 = 0(\%s) \textbf{ in} \\
 & \ldots \\
 & \textbf{let}\, x_n = 4n(\%s) \textbf{ in} \\
 & \textbf{let}\, r = [\![e_2[x_1,\ldots,x_n]]\!] \textbf{ in} \\
 & a[r]
\end{array}
$$

Tuple allocation

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

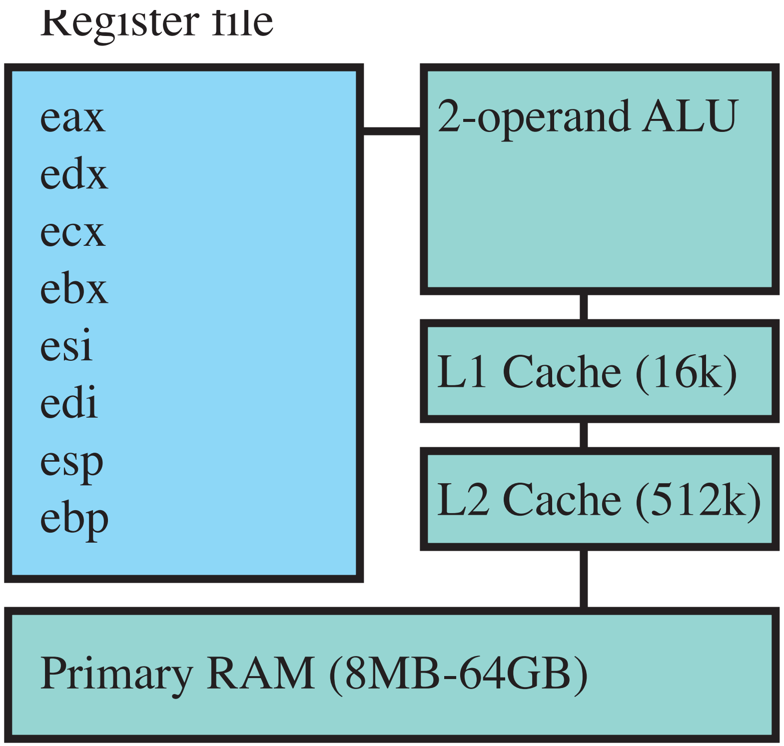
$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathbf{asm}\, r = \llbracket (e_1, \ldots, e_n) \rrbracket \;\mathbf{in}\; a[r]} \\
\longrightarrow & \mathbf{asm}\, r_1 = \llbracket e_1 \rrbracket \;\mathbf{in} \\
& \ldots \\
& \mathbf{asm}\, r_n = \llbracket e_n \rrbracket \;\mathbf{in} \\
& \mathbf{let}\, r = \mathbf{alloc}(\%r_1, \ldots, \%r_n) \;\mathbf{in} \qquad \#\ \text{Cheat!} \\
& a[r]
\end{array}
$$

Function call

$$
\begin{array}{l}
\textbf{asm}\, r = [\![e(e_1, \ldots, e_n)]\!] \textbf{ in } a[r] \\
\longrightarrow \textbf{asm}\, r_i = [\![e_i]\!] \textbf{ in} \\
\qquad \ldots \\
\qquad \textbf{asm}\, r_c = [\![e]\!] \textbf{ in} \\
\qquad \textbf{asm}\, r_f = 0(\%r_c) \textbf{ in} \qquad \text{\# Function pointer} \\
\qquad \textbf{let}\, r = (*\%r_f)(\%r_c, \%r_1, \ldots, \%r_n) \textbf{ in} \\
\qquad a[r]
\end{array}
$$

Step 2: register allocation

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



Register allocation

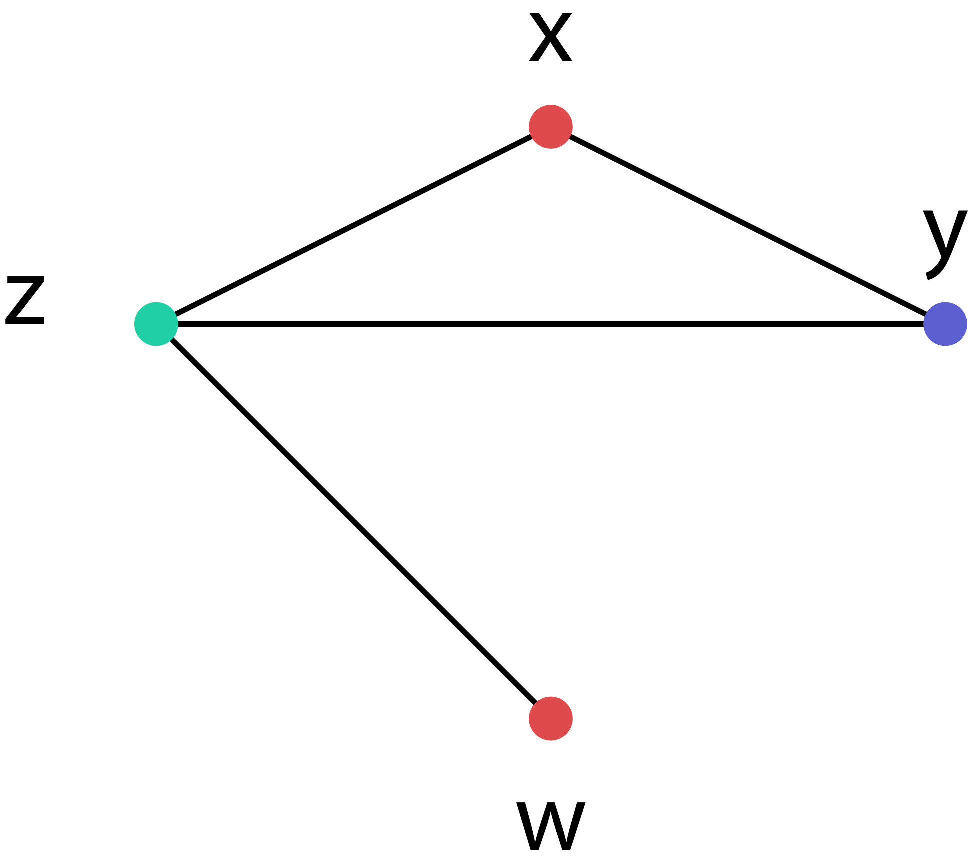
- Use α-renaming to use only register names for the variables
- There will be a *lot* of shadowing
- Formally, this is invisible!

$$\begin{array}{lcl}
\textbf{let}\ f(r_1, r_2) = & \rightarrow & \textbf{let}\ f(eax, ebx) = \\
\quad \textbf{let}\ r_3 = \%r_1 + \%r_2\ \textbf{in} & & \quad \textbf{let}\ eax = \%eax + \%ebx\ \textbf{in} \\
\quad \textbf{let}\ r_4 = \%r_3 + \$1\ \textbf{in} & & \quad \textbf{let}\ eax = \%eax + \$1\ \textbf{in} \\
\quad \%r_4 & & \quad \%eax
\end{array}$$

Chaitin-style graph coloring

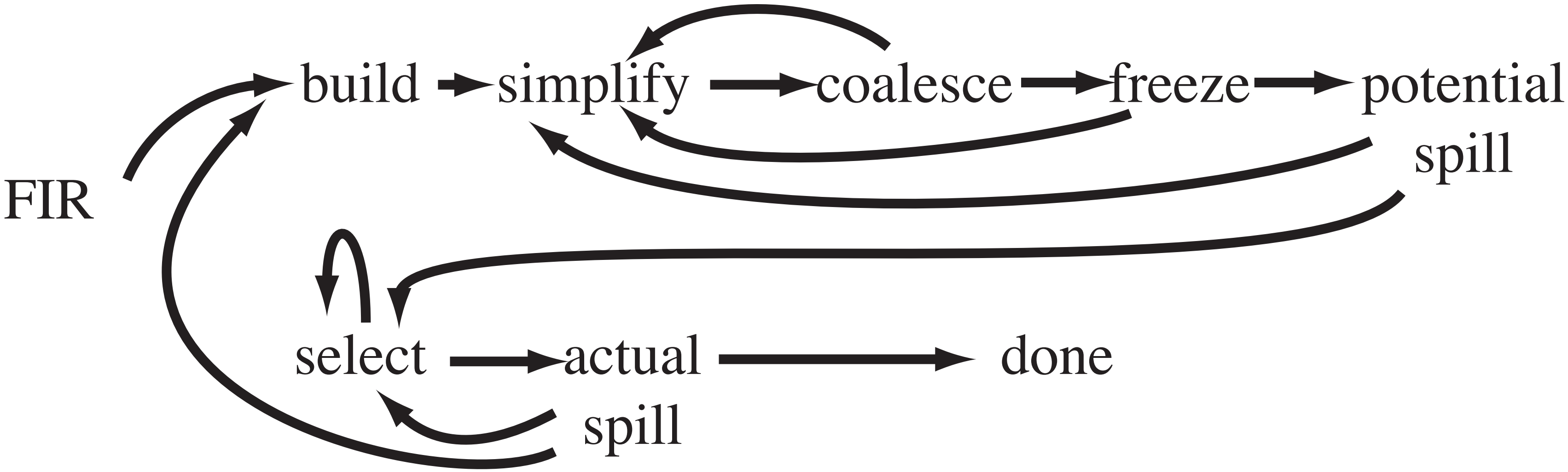
- Construct a graph with 1 node for each variable
- A variable is live from the point where it is defined, to the last point where it is used
- Two variables interfere iff they are both live at some program point
  - *Add an edge between interfering variables*
- Color the graph so adjacent vertices have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

Graph coloring



- ● %eax
- ● %ebx
- ● %ecx
- ● %edx
- ● %esi
- ● %edi

Algorithm



FIR

build

simplify

coalesce

freeze

potential spill

select

actual spill

done

Summer School on Logic and Theorem Proving in Programming Languages
http://www.cs.uoregon.edu/research/summerschool/summer08/

Formal Compilers
Jul 30, 2008

Spills

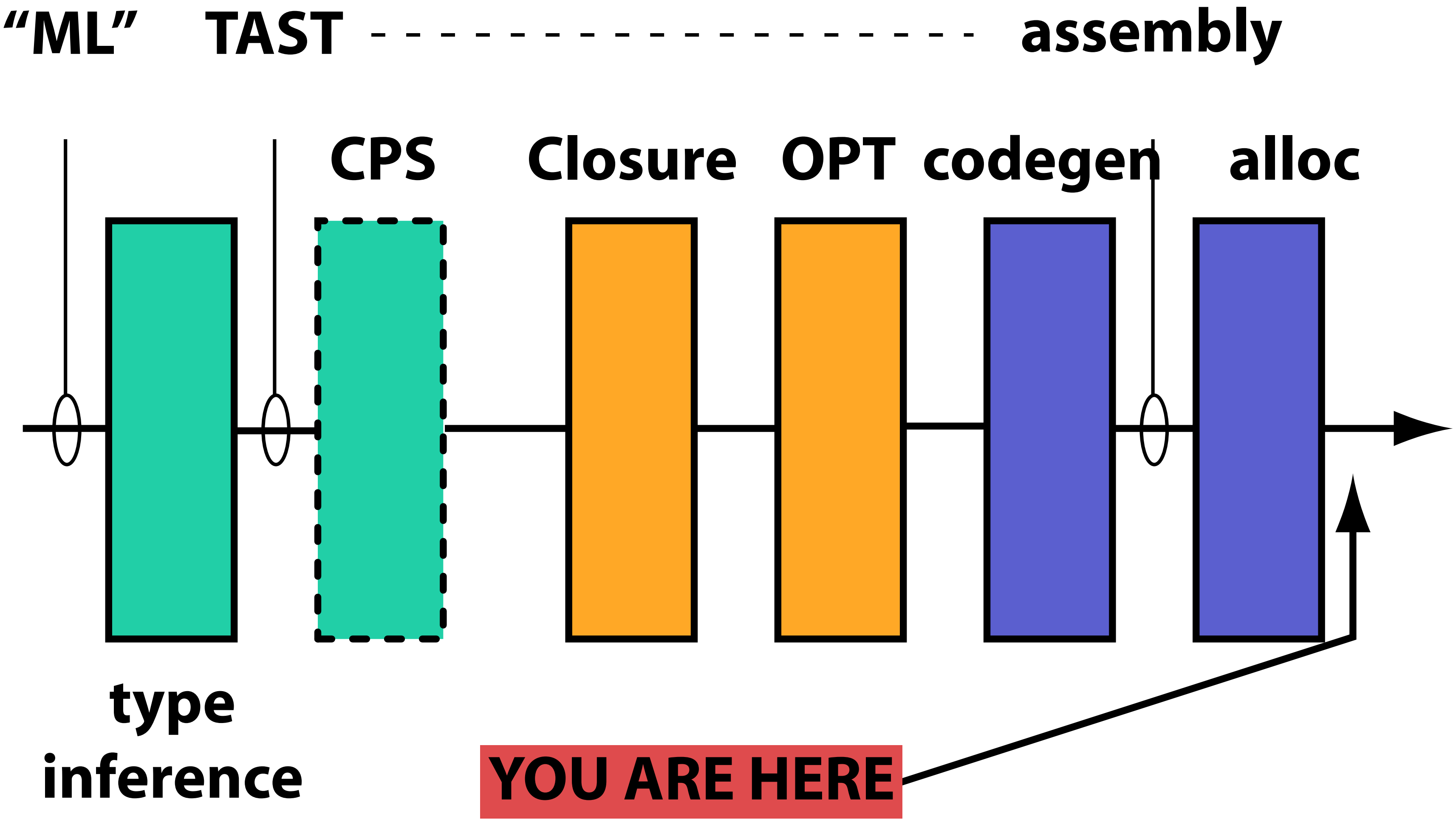
- Come back to reality!
- A real machine has a finite number of registers
  - *(6)*
- When too many variables are simultaneously live, some have to be "spilled": stored in memory

$$\begin{aligned}
&\textbf{let}\, r = e_1 \,\textbf{in}\, e_2[r] \\
\longrightarrow\ &\textbf{let}\, r = e_1 \,\textbf{in} \\
&\textbf{spill}\, s = r \,\textbf{in} \\
&e_2[s]
\end{aligned}$$

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
type inference
YOU ARE HERE

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*