CS153: Compilers
Lecture 13:
Compiling functions

Stephen Chong

https://www.seas.harvard.edu/courses/cs153
Contains content from lecture notes by Steve Zdancewic and Greg Morrisett
Mid-course Eval

Please rate your learning experience in the class so far
17 responses

- 58.8%: Excellent
- 23.5%: Good
- 11.8%: Fair
- 2%: 2
- 1%: Poor

• Most effective:
  - Homeworks
  - Lectures
  - Piazza/OH

• Least effective:
  - Looking at code in class (too much, too little!)
  - Lecture material not relevant to current assignment
  - OH for extension
Mid-course Eval

• Suggestions
  • Homework solutions
  • Idiomatic OCaml code
  • Go faster
  • “Stop using ocaml, it gets in the way of learning about compilers”, “This is not supposed to be an ocaml course, it's supposed to be a compilers course”
  • More type annotations in homework stub code
  • Long time to get OCaml set up
  • ...

Stephen Chong, Harvard University
Workload

- **Pie Chart:**
  - 52.9%: Too light
  - 41.2%: Just right

- **Bar Charts: HW1, HW2, HW3**
  - Columns represent hours spent on homework.
  - Bar heights indicate the number of students.

- **Legend:**
  - 1: Too light
  - 2
  - 3: Just right
  - 4
  - 5: Too heavy
Mid-course Eval: Actions

• Concrete actions course staff:
  • More type annotations in future HWs
  • Will release reference solutions
  • Lectures will be same pace or a bit faster (but will still have lots of time for questions)

• Concrete actions students:
  • Contact course staff re OH frequency/timing; we will try to adjust
  • Contact for additional info/feedback on graded HWs
  • Start HW early, reach out early and often for help

• Notes:
  • Implementation course: coding/coding style is important
  • Pedagogical decision to release HWs only after material is covered
Today

- Closure conversion
- Implementing environments and variables
  - DeBruijn indices
  - Nested environments vs flat environments
• Instead of doing substitution on nested functions when we reach the lambda, we can instead make a promise to finish the substitution if the nested function is ever applied.

• Instead of

\[
\text{Lambda}(x, e') \rightarrow \text{Lambda}(x, \text{subst env } e')
\]

we will have, in essence,

\[
\text{Lambda}(x, e') \rightarrow \text{Promise}(\text{env}, \text{Lambda}(x, e'))
\]

• Called a closure

• Need to modify rule for application to expect environment
Closure-based Semantics

```ocaml
type value = Int_v of int
    | Closure_v of {env:env, body:var*exp}
and env = (string * value) list

let rec eval (e:exp) (env:env) : value =
    match e with
    | Int i -> Int_v i
    | Var x -> lookup env x
    | Lambda(x,e) -> Closure_v{env=env, body=(x,e)}
    | App(e1,e2) ->
        (match eval e1 env, eval e2 env with
         | Closure_v{env=cenv, body=(x,e')}, v ->
            eval e' ((x,v)::cenv))
```
Inference rules

\[ \Gamma \vdash i \Downarrow i \\quad \Gamma \vdash i \Downarrow i \\quad \Gamma \vdash e_1 \Downarrow i_1 \quad \Gamma \vdash e_2 \Downarrow i_2 \quad i = i_1 + i_2 \]

\[ \Gamma \vdash \text{fun } x \to e \Downarrow (\Gamma, \text{fun } x \to e) \]

\[ \Gamma \vdash e_1 \Downarrow (\Gamma_c, \text{fun } x \to e) \quad \Gamma \vdash e_2 \Downarrow v \quad \Gamma_c[x \mapsto v] \vdash e \Downarrow w \]

\[ \Gamma \vdash e_1 \; e_2 \Downarrow w \]
So, How Do We Compile Closures?

- Represent function values (i.e., closures) as a pair of function pointer and environment
- Make all functions take environment as an additional argument
  - Access variables using environment
- Can then move all function declarations to top level (i.e., no more nested functions!)
- E.g., `fun x -> (fun y -> y+x)` becomes, in C-like code:

  ```c
  closure *f1(env *env, int x) {
    env *el = extend(env,"x",x);
    closure *c =
        malloc(sizeof(closure));
    c->env = el; c->fn = &f2;
    return c;
  }

  int f2(env *env, int y) {
    env *el = extend(env,"y",y);
    return lookup(el, "y") + lookup(el, "x");
  }
  ```
Where Do Variables Live

• Variables used in outer function may be needed for nested function
  • e.g., variable $x$ in example on previous slide
• So variables used by nested functions can’t live on stack...
• Allocate record for all variables on heap
• This will be similar to objects (which we will see in a few lectures)
  • Object = struct for field values, plus pointer(s) to methods
  • Closure = environment plus pointer to code
Closure Conversion

• Converting function values into closures
  • Make all functions take explicit environment argument
  • Represent function values as pairs of environments and lambda terms
  • Access variables via environment

• E.g.,

\[
\text{fun } x \rightarrow (\text{fun } y \rightarrow y+x) \\
\text{becomes} \\
\text{fun } \text{env} \ x \rightarrow \\
\quad \text{let } e' = \text{extend } \text{env} \ "x" \ x \ \text{in} \\
\quad (e', \ \text{fun } \text{env} \ y \rightarrow \\
\quad \quad \text{let } e' = \text{extend } \text{env} \ "y" \ y \ \text{in} \\
\quad \quad \quad (\text{lookup } e' \ "y")+(\text{lookup } e' \ "x"))
\]
Lambda Lifting

• After closure conversion, nested functions do not directly use variables from enclosing scope
• Can “lift” the lambda terms to top level functions!

E.g., \texttt{fun env \ x \ ->}
\begin{verbatim}
    let e' = extend env "x" x in
    (e', \texttt{fun env \ y \ ->}
        let e' = extend env "y" y in
        (lookup e' "y")+(lookup e' "x"))
\end{verbatim}

becomes
\begin{verbatim}
let f2 = \texttt{fun env \ y \ ->}
    let e' = extend env "y" y in
    (lookup e' "y")+(lookup e' "x")
\texttt{fun env \ x \ ->}
    let e' = extend env "x" x in
    (e', f2)
\end{verbatim}
Lambda Lifting

• E.g., \( \text{fun } \text{env } x \rightarrow \)

\[
\text{let } e' = \text{extend env } ''x'' \text{ x in } \\
(e', \text{ fun } \text{env } y \rightarrow \\
\text{let } e' = \text{extend env } ''y'' \text{ y in } \\
(\text{lookup } e' ''y'')+(\text{lookup } e' ''x''))
\]

becomes

\[
\text{let } f2 = \text{fun } \text{env } y \rightarrow \\
\text{let } e' = \text{extend env } ''y'' \text{ y in } \\
(\text{lookup } e' ''y'')+(\text{lookup } e' ''x'')
\]

\[
\text{fun } \text{env } x \rightarrow \\
\text{let } e' = \text{extend env } ''x'' \text{ x in } \\
(e', f2)
\]

closure *f1(env *env, int x) { 
    env *e1 = extend(env,"x",x);
    closure *c = 
        malloc(sizeof(closure));
    c->env = e1; c->fn = &f2;
    return c;
}

int f2(env *env, int y) {
    env *e1 = extend(env,"y",y);
    return lookup(e1,"y")
        + lookup(e1,"x");
}
How Do We Compile Closures Efficiently?

- Don’t need to heap allocate all variables
  - Just the ones that “escape”, i.e., might be used by nested functions
- Implementation of environment and variables
DeBruijn Indices

- In our interpreter, we represented environments as lists of pairs of variables names and values
- Expensive string comparison when looking up variable! `lookup env x`

```ocaml
let rec lookup env x =
    match env with
    | ((y,v)::rest) ->
        if y = x then v else lookup rest
    | [] -> error "unbound variable"
```

- Instead of using strings to represent variables, we can use natural numbers
  - Number indicates lexical depth of variable
DeBruijn Indices

\[
\text{type } \text{exp} = \text{Int of int | Var of int} \\
\quad | \text{Lambda of exp | App of exp*exp}
\]

• Original program

\[
\text{fun } x \rightarrow \text{fun } y \rightarrow \text{fun } z \rightarrow x + y + z
\]

• Conceptually, can rename program variables

\[
\text{fun } x2 \rightarrow \text{fun } x1 \rightarrow \text{fun } x0 \rightarrow x2 + x1 + x0
\]

• Don’t bother with variable names at all!

\[
\text{fun } \rightarrow \text{fun } \rightarrow \text{fun } \rightarrow \text{Var } 2 + \text{Var } 1 + \text{Var } 0
\]

• Number of variable indicates lexical depth, 0 is innermost binder
Converting to DeBruijn Indices

```
type exp = Int of int | Var of int
        | Lambda of exp | App of exp * exp

let rec cvt (e:exp) (env:var->int): D.exp =
match e with
| Int i -> D.Int i
| Var x -> D.Var (env x)
| App(e1,e2) ->
    D.App(cvt e1 env,cvt e2 env)
| Lambda(x,e) =>
    let new_env(y) =
        if y = x then 0 else (env y)+1
    in
    Lambda(cvt e new_env)
```
New Interpreter

type value = Int_v of int
         | Closure_v of {env:env, \textbf{body:exp}}

and \textbf{env} = \textbf{value list}

\textbf{let rec eval} (e:exp) (env:env) : value =
\textbf{match} e \textbf{with}
    | Int i -> Int_v i
    | Var x -> List.nth env x
    | Lambda e -> Closure_v{env=env, body=e}
    | App(e1,e2) ->
        (\textbf{match} eval e1 env, eval e2 env \textbf{with}
            | Closure_v{env=cenv, body=(x,e’)}, v ->
                eval e’ v::cenv)
Representing Environments

- Linked list (nested environments)
Representing Environments

(((fun \rightarrow fun \rightarrow fun \rightarrow Var \ 2 + Var \ 1 + Var \ 0) \ 21) \ 17)
Representing Environments

\[((\text{fun} \to \text{fun} \to \text{fun} \to \text{Var} \ 2 + \text{Var} \ 1 + \text{Var} \ 0) \ 21) \ 17) \ 4\]

- Linked list (nested environments)

\[\text{env} \]
Representing Environments

- Linked list (nested environments)
- Array (flat environment)

\[
((\text{fun} \to \text{fun} \to \text{fun} \to \text{Var} 2 + \text{Var} 1 + \text{Var} 0) \ 21) \ 17 \ 4
\]
Representing Environments

- Linked list (nested environments)
- Array (flat environment)
Representing Environments

- Linked list (nested environments)
- Array (flat environment)

```
(((\text{fun} \to \text{fun} \to \text{fun} \to \text{Var} 2 + \text{Var} 1 + \text{Var} 0) \, 21) \, 17) \, 4
```
Multiple Arguments

- Can extend DeBruijn indices to allow multiple arguments

\[
\text{fun } x \ y \ z \rightarrow \text{fun } m \ n \rightarrow x + z + n
\]

\[
\text{fun } \rightarrow \text{fun } \rightarrow \text{Var}(1,0) + \text{Var}(1,2) + \text{Var}(0,1)
\]

- Nested environments might then be

```
next  m  n
nil   x   y   z
```
Array-based Closures with N-ary Functions

\[(\text{fun } (x\ y\ z) \to (\text{fun } (m\ n) \to (\text{fun } p \to (\text{fun } q \to m + z)\ x))\]

Note how free variables are “addressed” relative to the closure due to shared env.

“follow 1 next ptr then look up index 0”

“follow 2 next ptrs then look up index 2”
Basic Architecture

Source Code → Parsing → Elaboration → Lowering → Optimization → Code Generation → Target Code

Front end

Back end
Elaboration

Untyped Abstract Syntax Trees

Typed Abstract Syntax Trees
Undefined Programs

• After parsing, we have AST
• We can interpret AST, or compile it and execute
• But: not all programs are well defined
  • E.g., 3/0, "hello" - 7, 42(19), using a variable that isn’t in scope, ...

• **Types** allow us to rule out many of these undefined behaviors
  • Types can be thought of as an approximation of a computation
  • E.g., if expression e has type int, then it means that e will evaluate to some integer value
  • E.g., we can ensure we never treat an integer value as if it were a function
Type Soundness

- Key idea: a well-typed program when executed does not attempt any undefined operation
- Make a model of the source language
  - i.e., an interpreter, or other semantics
  - This tells us which operations are partial
  - Partiality is different for different languages
    - E.g., “Hi” + “ world” and “na”*16 may be meaningful in some languages
- Construct a function to check types: \( tc : AST \rightarrow bool \)
  - AST includes types (or type annotations)
  - If \( tc \ e \) returns true, then interpreting \( e \) will not result in an undefined operation
- Prove that \( tc \) is correct
Simple Language

type tipe =
  Int_t
|  Arrow_t of tipe*tipe
|  Pair_t of tipe*tipe

type exp =
  Var of var | Int of int
|  Plus_i of exp*exp
|  Lambda of var * tipe * exp
|  App of exp*exp
|  Pair of exp * exp
|  Fst of exp | Snd of exp

Note: function arguments have type annotation
let rec interp (env:var->value)(e:exp) =
  match e with
  | Var x -> env x
  | Int i -> Int_v i
  | Plus_i(e1,e2) ->
    (match interp env e1, interp env e2 of
    | Int_v i, Int_v j -> Int_v(i+j)
    | _,_ -> failwith "Bad operands!"
    )
  | Lambda(x,t,e) -> Closure_v{env=env,code=(x,e)}
  | App(e1,e2) ->
    (match (interp env e1, interp env e2) with
    | Closure_v{env=cenv,code=(x,e)},v ->
      interp (extend cenv x v) e
    | _,_ -> failwith "Bad operands!"
    )
let rec tc (env: var->tipe) (e: exp) =
  match e with
  | Var x -> env x
  | Int _ -> Int_t
  | Plus_i(e1,e2) ->
    (match tc env e1, tc env e with
     | Int_t, Int_t -> Int_t
     | _,_ -> failwith "..." )
  | Lambda(x,t,e) -> Arrow_t(t, tc (extend env x t) e)
  | App(e1,e2) ->
    (match (tc env e1, tc env e2) with
     | Arrow_t(t1,t2), t ->
       if (t1 != t) then failwith "..." else t2
     | _,_ -> failwith "..."  )
• Type checker is almost like an approximation of the interpreter!
  • But interpreter evaluates function body only when function applied
  • Type checker always checks body of function
• We needed to assume the input of a function had some type $t_1$, and reflect this in type of function ($t_1 \rightarrow t_2$)
• At call site ($e_1 \ e_2$), we don’t know what closure $e_1$ will evaluate to, but can calculate type of $e_1$ and check that $e_2$ has type of argument
Growing the Language

- Adding booleans...

```ocaml
type tipe = ... | Bool_t

let rec interp env e = ...
| True -> True_v
| False -> False_v
| If(e1,e2,e3) -> (match interp env e1 with
  True_v -> interp env e2
  | False_v -> interp env e3
  | _ -> failwith "...")
```

Stephen Chong, Harvard University
let rec tc (env: var->tipe) (e: exp) =
  match e with
  ...
  | True -> Bool_t
  | False -> Bool_t
  | If(e1,e2,e3) ->
    (let (t1,t2,t3) = (tc env e1,tc env e2,tc env e3)
      in
      match t1 with
      | Bool_t ->
        if (t2 != t3) then error() else t2
      | _ -> failwith "..."
**Type Safety**

- “Well typed programs do not go wrong.”
  
  – Robin Milner, 1978

- **Note:** this is a **very** strong property.
  
  - Well-typed programs cannot “go wrong” by trying to execute undefined code (such as \( 3 + (\text{fun } x \rightarrow 2) \))
  
  - Simply-typed lambda calculus is guaranteed to terminate! (i.e. it isn't Turing complete)

- Depending on language, will not rule out **all** possible undefined behavior
  
  - E.g., 3/0, *NULL, ...
  
  - More sophisticated type systems can rule out more kinds of possible runtime errors
Judgements and Inference Rules

• We saw type checking algorithm in code
• Can express type-checking rules compactly and clearly using a type judgment and inference rules
Type Judgments

• In the judgment: \( E \vdash e : t \)
  • \( E \) is a typing environment or a type context
  • \( E \) maps variables to types. It is just a set of bindings of the form:
    \( x_1 : t_1, x_2 : t_2, \ldots, x_n : t_n \)
• If \( E \vdash e : t \) then expression \( e \) has type \( t \) under typing environment \( E \)
  • \( E \vdash e : t \) can be thought of as a set or relation
• For example:
  \[ x : \text{int}, b : \text{bool} \vdash \text{if (b) 3 else x : int} \]

• What do we need to know to decide whether “if (b) 3 else x” has type int in the environment \( x : \text{int}, b : \text{bool} \)?
  • \( b \) must be a bool  
    i.e.  \[ x : \text{int}, b : \text{bool} \vdash b : \text{bool} \]
  • 3 must be an int  
    i.e.  \[ x : \text{int}, b : \text{bool} \vdash 3 : \text{int} \]
  • \( x \) must be an int  
    i.e.  \[ x : \text{int}, b : \text{bool} \vdash x : \text{int} \]
Why Inference Rules?

• Compact, precise way of specifying language properties.
  • E.g. ~20 pages for full Java vs. 100’s of pages of prose Java Language Spec.

• Inference rules correspond closely to the recursive AST traversal that implements them

• Type checking (and type inference) is nothing more than attempting to prove a different judgment \( (E \vdash e : t) \) by searching backwards through the rules.

• Compiling in a context is nothing more than a collection of inference rules specifying yet a different judgment \( (G \vdash src \Rightarrow target) \)
  • Moreover, the compilation rules are very similar in structure to the typechecking rules

• Strong mathematical foundations
  • The “Curry-Howard correspondence”: Programming Language \sim Logic, Program \sim Proof, Type \sim Proposition
  • See CS152 if you’re interested in type systems!
Inference Rules

- For Oat, we will split environment $E$ into global variables $G$ and local variables $L$.
- Judgment $G; L \vdash e : t$ “expression $e$ is well typed and has type $t$”
- Judgment $G; L \vdash s$ “statement $s$ is well formed”

\[
\begin{align*}
\text{Premises} & : & G; L \vdash e : \text{bool} & \quad G; L \vdash s_1 & \quad G; L \vdash s_2 \\
\text{Conclusion} & \vdash & G; L \vdash \text{if } (e) s_1 \text{ else } s_2
\end{align*}
\]

- Equivalently: For any environment $G; L$, expression $e$, and statements $s_1$, $s_2$.

\[
G; L \vdash \text{if } (e) s_1 \text{ else } s_2
\]

holds if $G; L \vdash e : \text{bool}$ and $G; L \vdash s_1$ and $G; L \vdash s_2$ all hold.

- This rule can be used for any substitution of the syntactic metavariables $G$, $L$, $e$, $s_1$ and $s_2$. 
Simply-typed Lambda Calculus

- Note how these rules correspond to the code.
Type Checking Derivations

- A **derivation** or **proof tree** is a tree where nodes are instantiations of inference rules and edges connect a premise to a conclusion.
- Leaves of the tree are **axioms** (i.e. rules with no premises).
  - E.g., the INT rule is an axiom.
- Goal of the typechecker: verify that such a tree exists.
- Example: Find a tree for the following program using the inference rules on the previous slide:
  \[
  \vdash \text{(fun (x:int) -> x + 3)} 5 : \text{int}
  \]
• Note: the OCaml function typecheck verifies the existence of this tree. The structure of the recursive calls when running `tc` is same shape as this tree!

• Note that `x : int ∈ E` is implemented by the function `env`
Theorem: (simply typed lambda calculus with integers)

If \( \vdash e : t \) then there exists a value \( v \) such that \( e \downarrow v \).
Arrays

- Array constructs are not hard
- First: add a new type constructor: \( T[\] \)

\[
\begin{align*}
\text{NEW} & \quad E \vdash e_1 : \text{int} \quad E \vdash e_2 : T \\
& \quad E \vdash \text{new } T[e_1](e_2) : T[\] \\
\end{align*}
\]

\[
\begin{align*}
\text{INDEX} & \quad E \vdash e_1 : T[\] \quad E \vdash e_2 : \text{int} \\
& \quad E \vdash e_1[e_2] : T \\
\end{align*}
\]

\[
\begin{align*}
\text{UPDATE} & \quad E \vdash e_1 : T[\] \quad E \vdash e_2 : \text{int} \quad E \vdash e_3 : T \\
& \quad E \vdash e_1[e_2] = e_3 \text{ ok} \\
\end{align*}
\]

\( e_1 \) is the size of the newly allocated array. \( e_2 \) initializes the elements of the array.

Note: These rules don’t ensure that the array index is in bounds – that should be checked \textit{dynamically}. 
• ML-style tuples with statically known number of products:

• First: add a new type constructor: $T_1 \times \ldots \times T_n$

TUPLE

\[
E \vdash e_1 : T_1 \quad \ldots \quad E \vdash e_n : T_n \\
E \vdash (e_1, \ldots, e_n) : T_1 \times \ldots \times T_n
\]

PROJ

\[
E \vdash e : T_1 \times \ldots \times T_n \quad 1 \leq i \leq n \\
E \vdash \#i \ e : T_i
\]
• ML-style references (note that ML uses only expressions)
• First, add a new type constructor: T ref

\[
\frac{E \vdash e : T}{E \vdash \text{ref e} : T \text{ ref}}
\]

\[
\frac{E \vdash e \in T \text{ ref}}{E \vdash !e : T}
\]

\[
\frac{E \vdash e_1 \in T \text{ ref} \quad E \vdash e_2 : T}{E \vdash e_1 := e_2 : \text{ unit}}
\]

Note the similarity with the rules for arrays…
Oat Type Checking

• For HW5 we will add typechecking to Oat
  • And some other features

• XXX typing rules for Oat

• Example derivation

```javascript
var x1 = 0;
var x2 = x1 + x1;
x1 = x1 - x2;
return(x1);
```
Example Derivation

```
var x1 = 0;
var x2 = x1 + x1;
x1 = x1 - x2;
return(x1);
```

```
\begin{array}{cccc}
\mathcal{D}_1 & \mathcal{D}_2 & \mathcal{D}_3 & \mathcal{D}_4 \\
G_0; \cdot ; \text{int} \vdash & \text{var } x_1 = 0; \text{var } x_2 = x_1 + x_1; x_1 = x_1 - x_2; \text{return } x_1; \Rightarrow \cdot ; x_1: \text{int}, x_2: \text{int}
\end{array}
```

```
\vdash \text{var } x_1 = 0; \text{var } x_2 = x_1 + x_1; x_1 = x_1 - x_2; \text{return } x_1;
```
Example Derivation

\[ \mathcal{D}_1 = \frac{G_0; \vdash 0 : \text{int} \quad \text{[INT]}}{G_0; \vdash 0 : \text{int} \quad \text{[CONST]}} \]

\[ \frac{G_0; \vdash \text{var } x_1 = 0 \Rightarrow \cdot, x_1 : \text{int} \quad \text{[DECL]}}{G_0; \cdot; \text{int} \vdash \text{var } x_1 = 0; \Rightarrow \cdot, x_1 : \text{int} \quad \text{[SDECL]}} \]

\[ \mathcal{D}_2 = \frac{\vdash (+ : (\text{int}, \text{int}) \rightarrow \text{int}) \quad \text{[ADD]}}{\frac{x_1 : \text{int} \in \cdot, x_1 : \text{int} \quad \text{[VAR]}}{G_0; \cdot, x_1 : \text{int} \vdash x_1 : \text{int} \quad \text{[VAR]}}{G_0; \cdot, x_1 : \text{int} \vdash x_1 + x_1 : \text{int}} \quad \text{[BOP]} \]

\[ \frac{G_0; \cdot, x_1 : \text{int}; \text{int} \vdash \text{var } x_2 = x_1 + x_1; \Rightarrow \cdot, x_1 : \text{int}, x_2 : \text{int} \quad \text{[DECL]}}{G_0; \cdot, x_1 : \text{int}; \text{int} \vdash \text{var } x_2 = x_1 + x_1; \Rightarrow \cdot, x_1 : \text{int}, x_2 : \text{int} \quad \text{[SDECL]}} \]
Example Derivation

\[ D_3 \]

\[
\frac{x_1 : \text{int} \in \cdot, x_1 : \text{int}, x_2 : \text{int};}{\vdash - : (\text{int, int}) \rightarrow \text{int} \quad \text{[ADD]}} \quad \frac{x_1 : \text{int} \in \cdot, x_1 : \text{int}, x_2 : \text{int}}{G_0 ; \cdot, x_1 : \text{int}, x_2 : \text{int} \vdash x_1 : \text{int} \quad \text{[VAR]}} \quad \frac{x_2 : \text{int} \in \cdot, x_1 : \text{int}, x_2 : \text{int}}{G_0 ; \cdot, x_1 : \text{int}, x_2 : \text{int} \vdash x_2 : \text{int} \quad \text{[VAR]}} \quad \frac{}{G_0 ; \cdot, x_1 : \text{int}, x_2 : \text{int} \vdash x_1 - x_2 : \text{int} \quad \text{[BOP]}} \quad \frac{x_0 : \cdot, x_1 : \text{int}, x_2 : \text{int}; \text{int} \vdash x_1 = x_1 - x_2; \Rightarrow \cdot, x_1 : \text{int}, x_2 : \text{int}} {G_0 ; \cdot, x_1 : \text{int}, x_2 : \text{int}; \text{int} \vdash x_1 = x_1 - x_2; \Rightarrow \cdot, x_1 : \text{int}, x_2 : \text{int} \quad \text{[ASSN]}}
\]

\[ D_4 = \quad \frac{x_1 : \text{int} \in \cdot, x_1 : \text{int}, x_2 : \text{int}}{G_0 ; \cdot, x_1 : \text{int}, x_2 : \text{int} \vdash x_1 : \text{int} \quad \text{[VAR]}} \quad \frac{}{G_0 ; \cdot, x_1 : \text{int}, x_2 : \text{int}; \text{int} \vdash \text{return } x_1; \Rightarrow \cdot, x_1 : \text{int}, x_2 : \text{int} \quad \text{[RET]}}
\]
Type Safety For General Languages

**Theorem: (Type Safety)**

If \( \vdash P : t \) is a well-typed program, then either:

(a) the program terminates in a well-defined way, or
(b) the program continues computing forever

• Well-defined termination could include:
  • halting with a return value
  • raising an exception

• Type safety rules out undefined behaviors:
  • abusing “unsafe” casts: converting pointers to integers, etc.
  • treating non-code values as code (and vice-versa)
  • breaking the type abstractions of the language

• What is “defined” depends on the language semantics…
Compilation As Translating Judgments

- Consider the source typing judgment for source expressions:
  \[ C \vdash e : t \]

- How do we interpret this information in the target language?
  \[ \llbracket C \vdash e : t \rrbracket = ? \]

- \[ \llbracket C \rrbracket \] translates contexts
- \[ \llbracket t \rrbracket \] is a target type
- \[ \llbracket e \rrbracket \] translates to a (potentially empty) stream of instructions, that, when run, computes the result into some operand

- INARIANT: if \[ \llbracket C \vdash e : t \rrbracket = ty, \text{ operand}, \text{ stream} \] then the type (at the target level) of the operand is \( ty = \llbracket t \rrbracket \)
Example

- $C \leftarrow 37 + 5 : \text{int}$

What is $\llbracket C \leftarrow 37 + 5 : \text{int} \rrbracket$?

\[
\begin{align*}
\llbracket C \leftarrow 37 : \text{int} \rrbracket & = (\text{i64, Const } 37, []) \\
\llbracket C \leftarrow 5 : \text{int} \rrbracket & = (\text{i64, Const } 5, [])
\end{align*}
\]

\[
\begin{align*}
\llbracket C \leftarrow 37 : \text{int} \rrbracket & = (\text{i64, Const } 37, []) \\
\llbracket C \leftarrow 5 : \text{int} \rrbracket & = (\text{i64, Const } 5, [])
\end{align*}
\]

\[
\begin{align*}
\llbracket C \leftarrow 37 + 5 : \text{int} \rrbracket & = (\text{i64, } \%\text{tmp}, [\%\text{tmp} = \text{add i64 (Const } 37) (\text{Const } 5)])
\end{align*}
\]
What about the Context?

• What is $\llbracket C \rrbracket$?

• Source level C has bindings like: $x: \text{int}, y: \text{bool}$
  - We think of it as a finite map from identifiers to types

• What is the interpretation of C at the target level?

• $\llbracket C \rrbracket$ maps source identifiers, “x” to source types and $\llbracket x \rrbracket$

• What is the interpretation of a variable $\llbracket x \rrbracket$ at the target level?

  $x : t \in L$
  \[ TYP \ VAR \]  \[ \frac{}{G ; x : t \vdash \llbracket x \rrbracket} \]

  as expressions
  (which denote values)

• How are the variables used in the type system?

  $x : t \in L$
  \[ TYP \ ASSN \]  \[ \frac{G ; L \vdash \text{exp} : t}{\text{a} : x \vdash \text{eq} \llbracket \text{exp} \rrbracket} \]

  as addresses
  (which can be assigned)
Interpretation of Contexts

• $\llbracket C \rrbracket = \text{a map from source identifiers to types and target identifiers}$

• INVARIANT: 
  $x : t \in C$ means that 
  
  (1) $\text{lookup } \llbracket C \rrbracket x = (t, \%id_x)$ 
  (2) the (target) type of $\%id_x$ is $\llbracket t \rrbracket^*$ (a pointer to $\llbracket t \rrbracket$)
Interpretation of Variables

- Establish invariant for expressions:
  \[ \frac{G; L ⊢ x : t}{G; L ⊢ \text{TYP\_VAR}} \]
  as expressions
  (which denote values)

  \[ (\%\text{tmp}, [\%\text{tmp} = \text{load i64}\ast %\text{id}_x]) \]

  where \((\text{i64}, %\text{id}_x) = \text{lookup} [L] x\)

- What about statements?
  \[ \frac{x : t \in L \quad G; L ⊢ \text{exp} : t}{G; L; rt ⊢ x = \text{exp}; \Rightarrow L} \]
  as addresses
  (which can be assigned)

  \[ = \text{stream} @ \]
  \[ [\text{store} [t] \text{opn}, [t]\ast %\text{id}_x] \]

  where \((t, %\text{id}_x) = \text{lookup} [L] x\)
  and \([G;L \vdash \text{exp} : t] = ([t], \text{opn}, \text{stream})\)
Other Judgments?

• Statement:
\[ [C; rt \vdash stmt \Rightarrow C'] = [C'], \text{stream} \]

• Declaration:
\[ [G;L \vdash t x = \text{exp} \Rightarrow G;L,x:t ] = [G;L,x:t], \text{stream} \]

INVARIANT: stream is of the form:
\[
\text{stream' @} \\
[ \%id_x = \text{alloca} [t]; \\
\text{store} [t] \text{opn}, [t]* \%id_x ]
\]

and \[ [G;L \vdash \text{exp : t } ] = ([t], \text{opn}, \text{stream'}) \]

• Rest follow similarly
Compiling Control
Translating while

• Consider translating "while(e) s":
  • Test the conditional, if true jump to the body, else jump to the label after the body.

\[ \text{while}(e) \text{ s} \Rightarrow C' \] = [C'],

```c
lpre:
    opn = [C ⊢ e : bool]
    %test = icmp eq i1 opn, 0
    br %test, label %lpost, label %lbody

lbody:
    [C;rt ⊢ s ⇒ C']
    br %lpre

lpost:
```

• Note: writing \( \text{opn} = [C \leftarrow e : \text{bool}] \) is pun

  • translating \( [C \leftarrow e : \text{bool}] \) generates code that puts the result into \( \text{opn} \)
  • In this notation there is implicit collection of the code
Translating if-then-else

- Similar to while except that code is slightly more complicated because if-then-else must reach a merge.

```
opn = [C ⊢ e : bool]
%test = icmp eq i1 opn, 0
br %test, label %else, label %then
then:
  [C;rt ⊢ s₁ ⇒ C’]
  br %merge
else:
  [C; rt s₂ ⇒ C’]
  br %merge
merge:
```
Connecting this to Code

- **Instruction streams:**
  - Must include labels, terminators, and “hoisted” global constants

- **Must post-process the stream into a control-flow-graph**

- See `frontend.ml` from HW4