HARVARD
John A. Paulson
School of Engineering
and Applied Sciences

CS153: Compilers
Lecture 14: Type Checking

Stephen Chong

https://www.seas.harvard.edu/courses/cs153

Contains content from lecture notes by Steve Zdancewic and Greg Morrisett

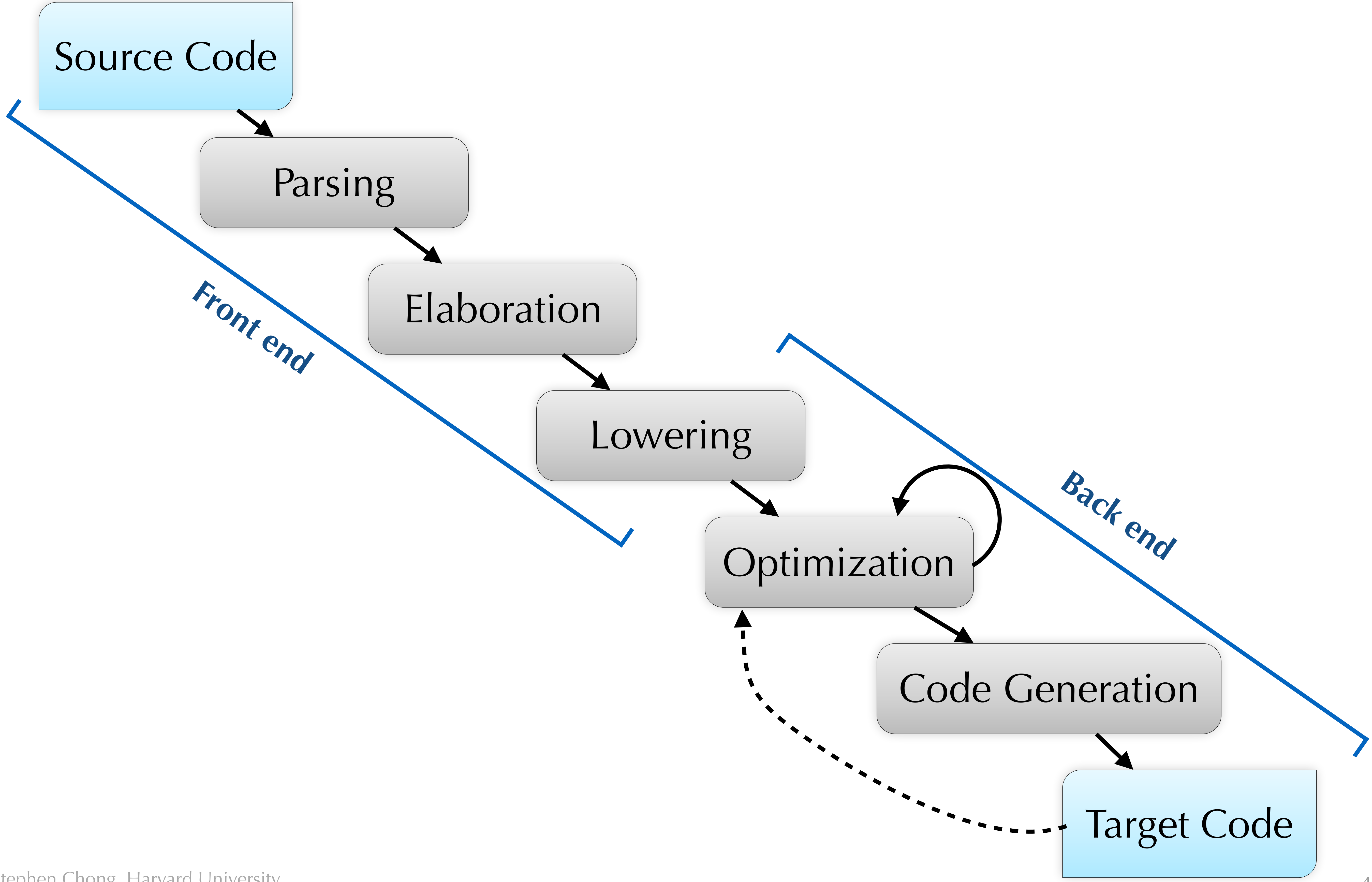
Announcements

- HW4 Oat v1 out
  - Due Tuesday Oct 29 (12 days)

Today

- Type checking
- Judgments and inference rules

Basic Architecture



Source Code

Parsing

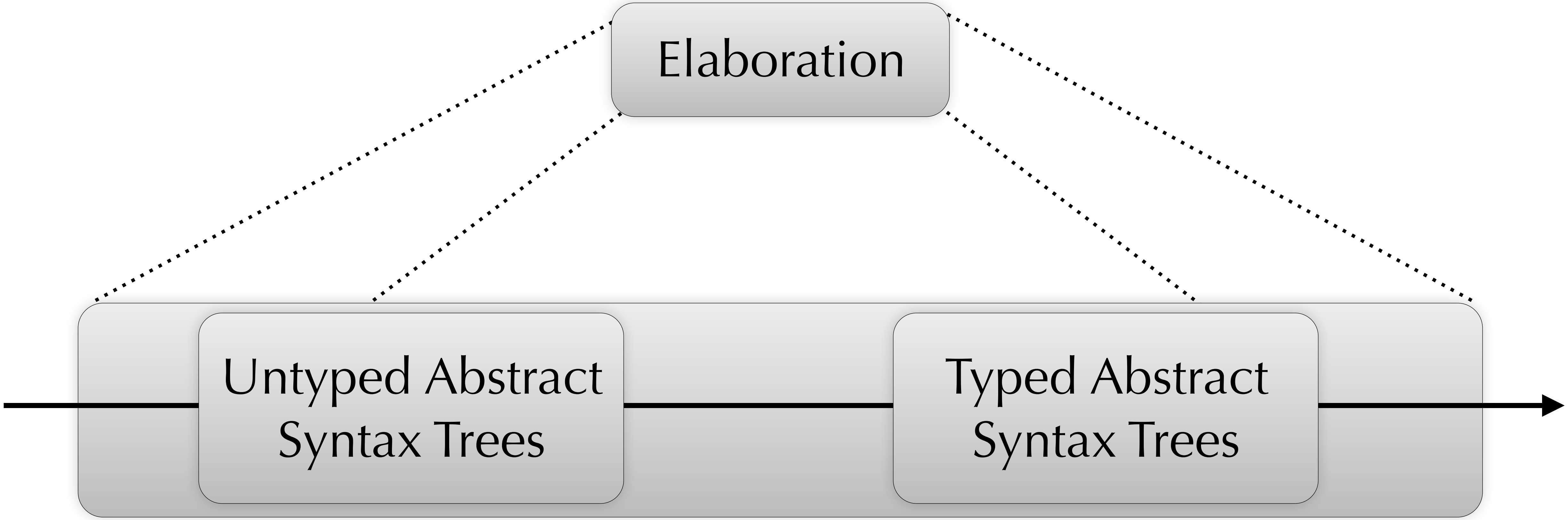
Elaboration

Lowering

Optimization

Code Generation

Target Code



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 -> 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

Interpreter

```
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!”)
```

Type Checker

```
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 “...”)
```

Notes

- 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 $\texttt{t}_1$, and reflect this in type of function ($\texttt{t}_1\texttt{->}\texttt{t}_2$)
- At call site ($\texttt{e}_1\ \ \texttt{e}_2$), we don't know what closure e_1 will evaluate to, but can calculate type of $\texttt{e}_1$ and check that $\texttt{e}_2$ has type of argument

Growing the Language

- Adding booleans...

```
type tipe = ... | Bool_t

type exp = ... | True | False | If of exp*exp*exp

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 “...”)
```

Type Checking

```
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 Inference

- Type checking is great if we already have enough type annotations
  - For our simple functional language, sufficient to have type annotations for function arguments
- But what about if we tried to infer types?
  - Reduce programmer burden!
- Efficient algorithms to do this: Hindley-Milner
  - Essentially build constraints based on how expressions are used and try to solve constraints
  - Error messages for non-well-typed programs can be challenging!

Polymorphism and Type Inference

- **Polymorphism** is the ability of code to be used on values of different types.
  - E.g., polymorphic function can be invoked with arguments of different types
  - Polymorph means “many forms”
- OCaml has polymorphic types
  - e.g., `val swap : 'a ref -> 'a -> ‘a = ...`
- But type inference for full polymorphic types is undecidable...
- OCaml has restricted form of polymorphism that allows type inference: **let-polymorphism** aka prenex polymorphism
  - Allow let expressions to be typed polymorphically, i.e., used at many types
  - Doesn’t require copying of let expressions
  - Requires clear distinction between polymorphic types and non-polymorphic types...

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 + (fun x -> 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

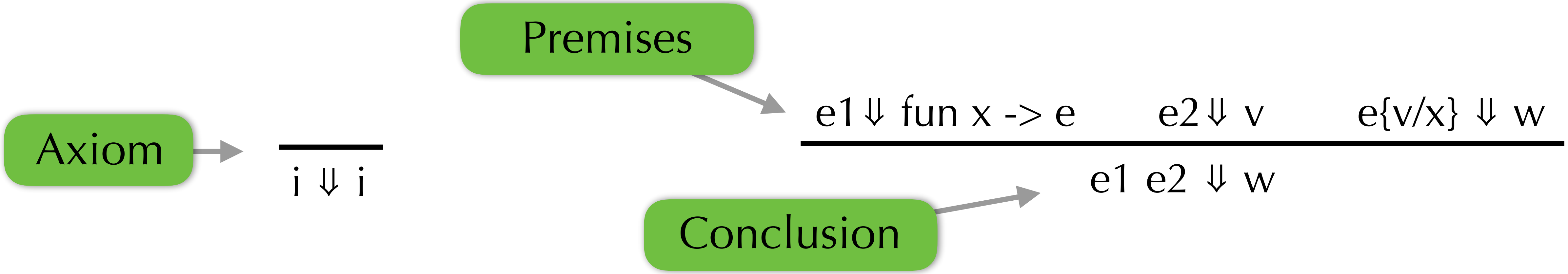
- We saw type checking algorithm in code
- Can express type-checking rules compactly and clearly using a **type judgment** and **inference rules**

Type Judgments

- In the judgment: E ⊢ 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: x1 : t1, x2 : t2, …, xn : tn
- If E ⊢ e : t then expression e has type t under typing environment E
  - E ⊢ e : t can be thought of as a set or relation
- For example:

  x : int, b : bool ⊢ 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 : int, b : bool?
  - b must be a bool i.e. x : int, b : bool ⊢ b : bool
  - 3 must be an int i.e. x : int, b : bool ⊢ 3 : int
  - x must be an int i.e. x : int, b : bool ⊢ x : int

Recall Inference Rules



Premises

Axiom

$$\frac{}{i \Downarrow i}$$

$$\frac{e1 \Downarrow \texttt{fun x -> e} \qquad e2 \Downarrow v \qquad e\{v/x\} \Downarrow w}{e1\ e2 \Downarrow w}$$

Conclusion

- Inference rule
  - If the premises are true, then the conclusion is true
  - An **axiom** is a rule with no premises
  - Inference rules can be **instantiated** by replacing **metavariables** (e, e1, e2, x, i, ...) with expressions, program variables, integers, as appropriate.

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 ($E \vdash src \Rightarrow target$)
  - Moreover, the compilation rules are very similar in structure to the typechecking rules
- Strong mathematical foundations
  - The "Curry-Howard correspondence": Programming Language ~ Logic, Program ~ Proof, Type ~ Proposition
  - See CS152 if you're interested in type systems!

Simply-typed Lambda Calculus

INT

$$\frac{}{E \vdash i : \text{int}}$$

VAR

$$\frac{x : T \in E}{E \vdash x : T}$$

ADD

$$\frac{E \vdash e_1 : \text{int} \qquad E \vdash e_2 : \text{int}}{E \vdash e_1 + e_2 : \text{int}}$$

FUN

$$\frac{E, x : T \vdash e : S}{E \vdash \text{fun } (x{:}T) \text{ -> } e \; : T \text{ -> } S}$$

APP

$$\frac{E \vdash e_1 : T \text{ -> } S \qquad E \vdash e_2 : T}{E \vdash e_1 \; e_2 : S}$$

- 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)
- 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:

$$\vdash \texttt{(fun (x:int) -> x + 3) 5} \ : \texttt{int}$$

Example Derivation Tree

$$\boxed{\text{APP}}\;\dfrac{\boxed{\text{FUN}}\;\dfrac{\boxed{\text{ADD}}\;\dfrac{\boxed{\text{VAR}}\;\dfrac{x : \text{int} \;\in\; x : \text{int}}{x : \text{int} \vdash x \;:\; \text{int}} \qquad \boxed{\text{INT}}\;\dfrac{}{x : \text{int} \vdash 3 \;:\; \text{int}}}{x : \text{int} \vdash x + 3 : \text{int}}}{\vdash (\text{fun } (x{:}\text{int}) \text{ -> } x + 3) : \text{int -> int}} \qquad \boxed{\text{INT}}\;\dfrac{}{\vdash 5 : \text{int}}}{\vdash (\text{fun } (x{:}\text{int}) \text{ -> } x + 3)\; 5 \;:\; \text{int}}$$

$$\boxed{\text{INT}}\;\dfrac{}{E \vdash i : \text{int}}$$

$$\boxed{\text{VAR}}\;\dfrac{x : T \;\in\; E}{E \vdash x : T}$$

$$\boxed{\text{ADD}}\;\dfrac{E \vdash e_1 : \text{int} \qquad E \vdash e_2 : \text{int}}{E \vdash e_1 + e_2 : \text{int}}$$

$$\boxed{\text{FUN}}\;\dfrac{E, x : T \vdash e : S}{E \vdash \text{fun } (x{:}T) \text{ -> } e \;:\; T \text{ -> } S}$$

$$\boxed{\text{APP}}\;\dfrac{E \vdash e_1 : T \text{ -> } S \qquad\qquad E \vdash e_2 : T}{E \vdash e_1\, e_2 : S}$$

Example Derivation Tree

$$
\text{APP}\;\dfrac{\text{FUN}\;\dfrac{\text{ADD}\;\dfrac{\text{VAR}\;\dfrac{x : \text{int} \;\in\; x : \text{int}}{x : \text{int} \vdash x \;: \text{int}} \qquad \text{INT}\;\dfrac{}{x : \text{int} \vdash 3 \;: \text{int}}}{x : \text{int} \vdash x + 3 : \text{int}}}{\vdash (\text{fun } (x{:}\text{int}) \text{ -> } x + 3) : \text{int -> int}} \qquad \text{INT}\;\dfrac{}{\vdash 5 : \text{int}}}{\vdash (\text{fun } (x{:}\text{int}) \text{ -> } x + 3)\ 5 \;: \text{int}}
$$

- 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 $x : \text{int} \in E$ is implemented by the function `lookup`

Type Safety Revisited

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[]

NEW

$$\frac{E \vdash e_1 : \text{int} \qquad E \vdash e_2 : T}{E \vdash \text{new } T[e_1](e_2) \ : T[]}$$

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

INDEX

$$\frac{E \vdash e_1 : T[] \qquad E \vdash e_2 : \text{int}}{E \vdash e_1[e_2] \ : T}$$

Note: These rules don't ensure that the array index is in bounds – that should be checked *dynamically*.

UPDATE

$$\frac{E \vdash e_1 : T[] \qquad E \vdash e_2 : \text{int} \qquad E \vdash e_3 : T}{E \vdash e_1[e_2] = e_3 \text{ ok}}$$

Tuples

- ML-style tuples with statically known number of products
- First: add a new type constructor: $T_1 * \ldots * T_n$

TUPLE

$$\frac{E \vdash e_1 : T_1 \quad \ldots \quad E \vdash e_n : T_n}{E \vdash (e_1, \ldots, e_n) : T_1 * \ldots * T_n}$$

PROJ

$$\frac{E \vdash e : T_1 * \ldots * T_n \quad 1 \leq i \leq n}{E \vdash \#i\ e\ :\ T_i}$$

References

- ML-style references (note that ML uses only expressions)
- First, add a new type constructor: T ref

REF

$$\frac{E \vdash e : T}{E \vdash \text{ref } e : T \text{ ref}}$$

DEREF

$$\frac{E \vdash e : T \text{ ref}}{E \vdash \,!e \; : T}$$

ASSIGN

$$\frac{E \vdash e_1 : T \text{ ref} \qquad E \vdash e_2 : T}{E \vdash e_1 := e_2 \; : \text{unit}}$$

Note the similarity with the rules for arrays…

Oat Type Checking

- For HW5 we will add typechecking to Oat
  - And some other features
- Some of Oat’s features
  - Imperative (update variables, like references)
  - Distinction between statements and expressions
  - More complicated control flow
    - Return
    - While, For, ...
- What does a type system look like for Oat?

Some Oat Judgments

- Split environment E into Globals and Locals
- Expression e has type t under context G;L
  - $G;\, L \vdash e : t$
- Statement s is well typed under context G;L. If it returns, it returns a value of type rt. After s, the local context is L′.
  - $G;\, L;\, rt \vdash s \Rightarrow L'$
- Where does G come from?
- Program is a list of global variable declarations and function declarations
- Use judgment to gather up global variable declarations
  - $\vdash_g prog \Rightarrow G$

Example Derivation

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

$$\dfrac{\dfrac{\mathcal{D}_1 \quad \mathcal{D}_2 \quad \mathcal{D}_3 \quad \mathcal{D}_4}{G_0\,;\,\cdot\,;\texttt{int} \vdash \texttt{var}\ x_1 = 0;\ \texttt{var}\ x_2 = x_1 + x_1\,;\ x_1 = x_1 - x_2\,;\ \texttt{return}\ x_1\,;\ \Rightarrow\ \cdot, x_1\!:\!\texttt{int}, x_2\!:\!\texttt{int}}\ [\textsc{stmts}]}{\vdash\ \texttt{var}\ x_1 = 0;\ \texttt{var}\ x_2 = x_1 + x_1\,;\ x_1 = x_1 - x_2\,;\ \texttt{return}\ x_1\,;}\ [\textsc{prog}]$$

Example Derivation

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

$$\mathcal{D}_1 \quad = \quad \dfrac{\dfrac{\dfrac{\dfrac{}{G_0\,;\cdot \vdash 0 : \mathtt{int}}\,[\text{INT}]}{G_0\,;\cdot \vdash 0 : \mathtt{int}}\,[\text{CONST}]}{G_0\,;\cdot \vdash \mathtt{var}\ x_1 = 0 \Rightarrow \cdot, x_1 : \mathtt{int}}\,[\text{DECL}]}{G_0\,;\,\cdot\,;\mathtt{int} \vdash \mathtt{var}\ x_1 = 0; \Rightarrow \cdot, x_1 : \mathtt{int}}\,[\text{SDECL}]$$

$$\mathcal{D}_2 \quad = \quad \dfrac{\dfrac{\dfrac{\dfrac{}{\vdash + : (\mathtt{int},\mathtt{int}) \rightarrow \mathtt{int}}\,[\text{ADD}] \quad \dfrac{x_1 : \mathtt{int} \in \cdot, x_1 : \mathtt{int}}{G_0\,;\cdot, x_1 : \mathtt{int} \vdash x_1 : \mathtt{int}}\,[\text{VAR}] \quad \dfrac{x_1 : \mathtt{int} \in \cdot, x_1 : \mathtt{int}}{G_0\,;\cdot, x_1 : \mathtt{int} \vdash x_1 : \mathtt{int}}\,[\text{VAR}]}{G_0\,;\cdot, x_1 : \mathtt{int} \vdash x_1 + x_1 : \mathtt{int}}\,[\text{BOP}]}{G_0\,;\cdot, x_1 : \mathtt{int}\,;\mathtt{int} \vdash \mathtt{var}\ x_2 = x_1 + x_1\,; \Rightarrow \cdot, x_1 : \mathtt{int}, x_2 : \mathtt{int}}\,[\text{DECL}]}{G_0\,;\cdot, x_1 : \mathtt{int}\,;\mathtt{int} \vdash \mathtt{var}\ x_2 = x_1 + x_1\,; \Rightarrow \cdot, x_1 : \mathtt{int}, x_2 : \mathtt{int}}\,[\text{SDECL}]$$

Example Derivation

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

$$\mathcal{D}_3 \quad = \quad \dfrac{\dfrac{\dfrac{}{\vdash \texttt{-} : (\texttt{int},\texttt{int}) \to \texttt{int}}[\textsc{add}] \quad \dfrac{x_1\!:\!\texttt{int} \in \cdot, x_1\!:\!\texttt{int}, x_2\!:\!\texttt{int}}{G_0\,;\cdot, x_1\!:\!\texttt{int}, x_2\!:\!\texttt{int} \vdash x_1 : \texttt{int}}[\textsc{var}] \quad \dfrac{x_2\!:\!\texttt{int} \in \cdot, x_1\!:\!\texttt{int}, x_2\!:\!\texttt{int}}{G_0\,;\cdot, x_1\!:\!\texttt{int}, x_2\!:\!\texttt{int} \vdash x_2 : \texttt{int}}[\textsc{var}]}{G_0\,;\cdot, x_1\!:\!\texttt{int}, x_2\!:\!\texttt{int} \vdash x_1 \,\texttt{-}\, x_2 : \texttt{int}}[\textsc{bop}]}{G_0\,;\cdot, x_1\!:\!\texttt{int}, x_2\!:\!\texttt{int}\,;\texttt{int} \vdash x_1 \,\texttt{=}\, x_1 \,\texttt{-}\, x_2\,\texttt{;} \Rightarrow \cdot, x_1\!:\!\texttt{int}, x_2\!:\!\texttt{int}}[\textsc{assn}]$$

$$\mathcal{D}_4 \quad = \quad \dfrac{\dfrac{x_1\!:\!\texttt{int} \in \cdot, x_1\!:\!\texttt{int}, x_2\!:\!\texttt{int}}{G_0\,;\cdot, x_1\!:\!\texttt{int}, x_2\!:\!\texttt{int} \vdash x_1 : \texttt{int}}[\textsc{var}]}{G_0\,;\cdot, x_1\!:\!\texttt{int}, x_2\!:\!\texttt{int}\,;\texttt{int} \vdash \texttt{return}\; x_1\,\texttt{;} \Rightarrow \cdot, x_1\!:\!\texttt{int}, x_2\!:\!\texttt{int}}[\textsc{Ret}]$$

Type Safety For General Languages

Theorem: (Type Safety)

If P 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?

$$[\![C \vdash e : t]\!] = \quad ?$$

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

- INVARIANT: if $[\![C \vdash e : t]\!]$ = ty, operand, stream
  then the type (at the target level) of the operand is ty=$[\![t]\!]$

Example

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

$$\dfrac{\dfrac{}{\llbracket C \vdash 37 : \text{int} \rrbracket = \texttt{(i64, Const 37, [])}} \qquad \dfrac{}{\llbracket C \vdash 5 : \text{int} \rrbracket = \texttt{(i64, Const 5, [])}}}{\llbracket C \vdash 37 + 5 : \text{int} \rrbracket = \texttt{(i64, \%tmp, [\%tmp = add i64 (Const 37) (Const 5)])}}$$

What about the Context?

- What is ⟦C⟧?
- Source level C has bindings like: x:int, y:bool
  - We think of it as a finite map from identifiers to types
- What is the interpretation of C at the target level?
- ⟦C⟧ maps source identifiers, "x", to target types and ⟦x⟧
- What is the interpretation of a variable ⟦x⟧ at the target level?
  - How are the variables used in the type system?

$$\frac{x : t \in L}{G; L \vdash x : t}\text{TYP\_VAR}$$

as expressions
(which denote values)

$$\frac{x : t \in L \quad G; L \vdash exp : t}{G; L; rt \vdash x = exp; \Rightarrow L}\text{TYP\_ASSN}$$

as addresses
(which can be assigned)

Interpretation of Contexts

- $[\![C]\!]$ = a map from source identifiers to types and target identifiers
- INVARIANT:

  $x{:}t \in C$ means that

  (1) lookup $[\![C]\!]$ x = ($[\![t]\!]$*, `%id_x`)

  (2) the (target) type of `%id_x` is $[\![t]\!]$* (a pointer to $[\![t]\!]$)

Interpretation of Variables

- Establish invariant for expressions:

$$\left[\!\!\left[\begin{array}{c}\dfrac{x : t \in L}{G; L \vdash x : t}\,\text{TYP\_VAR}\\ \text{as expressions}\\ \text{(which denote values)}\end{array}\right]\!\!\right]$$

= `(%tmp, [%tmp = load i64* %id_x])`

where (`i64`, `%id_x`) = lookup ⟦L⟧ x

- What about statements?

$$\left[\!\!\left[\begin{array}{c}\dfrac{x : t \in L \quad G; L \vdash exp : t}{G; L; rt \vdash x = exp; \Rightarrow L}\,\text{TYP\_ASSN}\\ \text{as addresses}\\ \text{(which can be assigned)}\end{array}\right]\!\!\right]$$

= stream `@ [store ⟦t⟧ opn, ⟦t⟧* %id_x]`

where (⟦`t`⟧, `%id_x`) = lookup ⟦L⟧ x
and ⟦G;L ⊢ exp : t⟧ = (⟦t⟧, `opn`, stream)