

CIS 341: Compilers

Lecture 17

The Plan

- Today:
 - Static analysis: Type checking
- Project 3: Code generation for the “T” control-transfer language
 - Due: Today at 11:59 pm
- Announcement:
 - Midterm is Monday, Oct. 20th in class.
 - Prof. Zdancewic will be out of town

Type Soundness

- Build a model of the programming language
 - One model: an interpreter
 - Another model: constructed in mathematics
 - Usually defined via the abstract syntax
- Model defines where the language operations are partial
 - Partiality is different for different languages: e.g. “foo” + “bar” is meaningful in Java but not OCaml
- Construct a function: **well_typed : Ast.t -> bool**
 - When **well_typed e = true**, running e should not result in an undefined operation.
- Prove that the **well_typed** function is correct.
 - Such proofs are sometimes difficult, but doable for real languages (e.g. SML, Java)

Typechecking

- How do we implement the function `well_typed`?
- Big idea: “approximate” the interpreter:
 - Problem is partiality in the language semantics as defined by the interpreter.
 - Instead of interpreting the program, write a function called **typecheck** that computes a type for the program (rather than the answer obtained by running the program).
 - Behavior of **typecheck** is guided by what the interpreter would do.
- See “tc.ml”

Notes about this Typechecker

- In the interpreter, we only evaluate the body of a function when it's applied.
- In the typechecker, we always check the body of the function (even if it's never applied.)
- Because of this, we must *assume* the input has some type (say t_1) and reflect this in the type of the function ($t_1 \rightarrow t_2$).
- Dually, at a call site ($e_1 e_2$), we don't know what *closure* we're going to get.
- But we can calculate e_1 's type, check that e_2 is an argument of the right type, and also determine what type e_1 will return.
- Question: Why is this an approximation?
- Question: What if **well_typed** always returns **false**?

Defining Type Systems Mathematically

- In the OCaml implementation we have:
typecheck (env:environment) (e:exp):ty
 - Where **exp** is the type of abstract syntax and **environment** is a **list** of **var * ty** pairs.
 - The result of **typecheck** is a type
- We can abstract this function in math as a relation:
The notation: $E \vdash e : t$ means $\text{typecheck } C e = t$
 - “In the environment E , program e is well-typed and has type t ”
 - “ $e : t$ ” is a *type judgment*
- Simple examples: $\vdash 3 : \text{int}$ $\vdash \text{true} : \text{bool}$ $\vdash \text{“hello”} : \text{string}$
- Bigger examples: $\vdash (2 * 3) + 5 : \text{int}$ $\vdash \text{if (true) 3 else 4} : \text{int}$

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:
 $x_1 : t_1, x_2 : t_2, \dots, x_n : t_n$
- For example: $x : \text{int}, b : \text{bool} \vdash \text{if } (b) \ 3 \ \text{else } x : \text{int}$
- 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. $x : \text{int}, b : \text{bool} \vdash x : \text{int}$

Generalizing 'if' & Inference Rules

- For any environment E , expressions e_1, e_2, e_3 , and type T the judgment

$$E \vdash \text{if } (e_1) e_2 \text{ else } e_3 : T$$

is true if $E \vdash e_1 : \text{bool}$, and $E \vdash e_2 : T$, and $E \vdash e_3 : T$ are all true.

- More succinctly: we summarize this as an *inference rule*:

$$\begin{array}{l} \text{Premises} \left\{ \begin{array}{l} E \vdash e_1 : \text{bool} \quad E \vdash e_2 : T \quad E \vdash e_3 : T \\ \hline E \vdash \text{if } (e_1) e_2 \text{ else } e_3 : T \end{array} \right. \\ \text{Conclusion} \left\{ \end{array}$$

- This rule holds for *any* substitution of the syntactic metavariables E , e_1 , e_2 , e_3 , and T

Simply-typed Lambda Calculus

- For the language in “tc.ml” we have five inference rules:

| | | |
|--|--|---|
| INT | VAR | ADD |
| $\frac{}{E \vdash i : \text{int}}$ | $\frac{x : T \in E}{E \vdash x : T}$ | $\frac{E \vdash e_1 : \text{int} \quad E \vdash e_2 : \text{int}}{E \vdash e_1 + e_2 : \text{int}}$ |
| FUN | APP | |
| $\frac{E, x : T \vdash e : S}{E \vdash \text{fun } (x:T) \rightarrow e : T \rightarrow S}$ | $\frac{E \vdash e_1 : T \rightarrow S \quad E \vdash e_2 : T}{E \vdash e_1 e_2 : S}$ | |

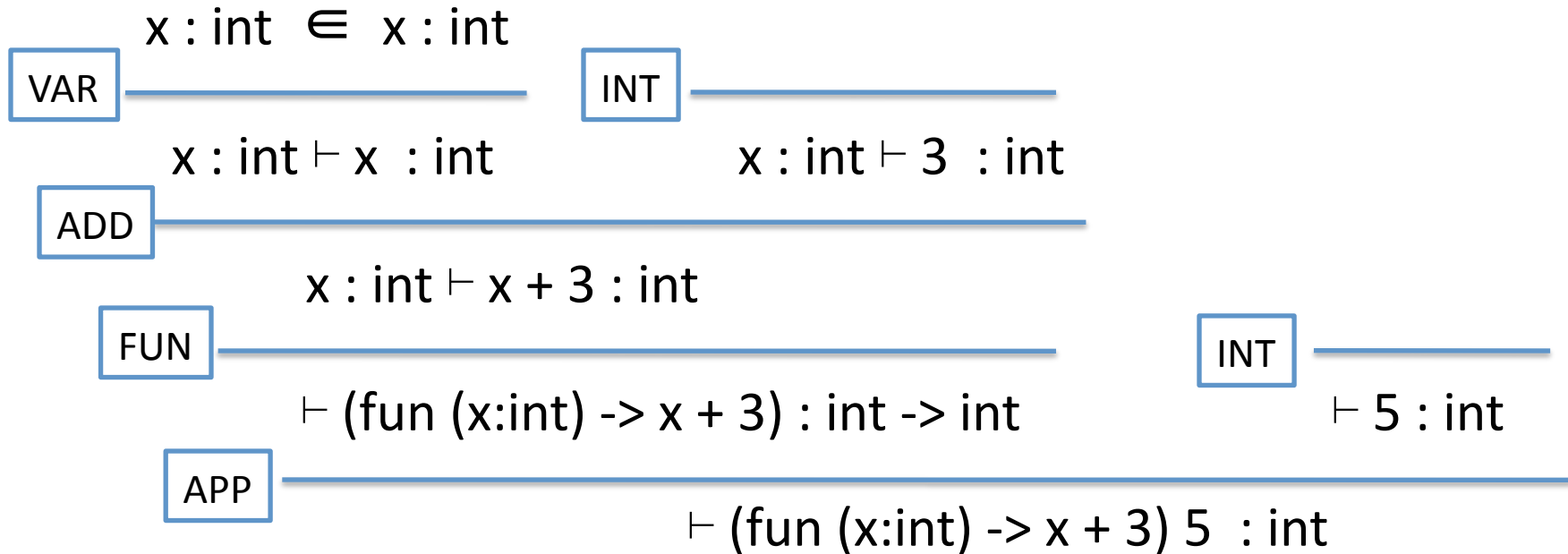
- Note how these rules correspond to the code.

Type Checking Derivations

- A *derivation* or *proof tree* has (instances of) judgments as its nodes and edges that connect premises to a conclusion according to an inference rule.
- Leaves of the tree are *axioms* (i.e. rules with no premises)
 - Example: 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:

$\vdash (\text{fun } (x:\text{int}) \rightarrow x + 3) 5 : \text{int}$

Example Derivation Tree



- Note: the OCaml function **typecheck** verifies the existence of this tree. The structure of the recursive calls when running **typecheck** is the same shape as this tree!
- Note that $x : \text{int} \in E$ is implemented by the function **lookup**

Why Inference Rules?

- They are a compact, precise way of specifying a type system.
 - E.g. ~20 pages for full Java vs. 100's of pages of 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 $E \vdash e : T$ by searching backwards through the rules.
- Strong mathematical foundations
 - The “Curry-Howard” isomorphism: Programming Language \sim Logic, Program \sim Proof, Type \sim Proposition
 - See CIS 500 next Spring if you're interested in type systems!