

CIS 341: Compilers

Lecture 18

The Plan

- Today:
 - Type checking continued
- Project 4: Code generation & typechecking for the “AT” (arrays and types) language
 - Due: October 24th at 11:59 pm
- Announcement:
 - Midterm is Monday, Oct. 20th in class.
 - Prof. Zdancewic will be out of town

CIS 341: Compilers2

Recap: Inference Rules

- Last time, we saw how a type system can be concisely specified using *inference rules*.

Premises $\left\{ \begin{array}{l} E \vdash e_1 : \text{bool} \quad E \vdash e_2 : T \quad E \vdash e_3 : T \end{array} \right.$

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

- The goal of the type checker is to find a derivation tree that establishes that the given program is well-typed.
- A language is *type safe* if whenever $\vdash e : T$ can be derived evaluating e results in no use of a partially defined function.
 - As a corollary: If e terminates with a value v , we have: $\vdash v : T$.
 - This should hold for all programs e .

CIS 341: Compilers3

Example Derivation Tree

VAR $\frac{x : \text{int} \in x : \text{int}}{x : \text{int} \vdash x : \text{int}}$ **INT** $\frac{}{x : \text{int} \vdash 3 : \text{int}}$

ADD $\frac{x : \text{int} \vdash x : \text{int} \quad x : \text{int} \vdash 3 : \text{int}}{x : \text{int} \vdash x + 3 : \text{int}}$

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

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

- 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**

CIS 341: Compilers4

Adding More Typing Rules

- It is easy to add inference rules for other program constructs:

WHILE $\frac{E \vdash e_1 : \text{int} \quad E \vdash e_2 : T}{E \vdash \text{while } (e_1) e_2 : \text{unit}}$ Note: If the language has Booleans, we should require: $E \vdash e_1 : \text{bool}$.

LET $\frac{E \vdash e_1 : T \quad E, x : T \vdash e_2 : S}{E \vdash T x = e_1; e_2 : S}$ Note: We add the assumption $x : T$ to the context when checking e_2 – x is in scope in e_2 .

ASSIGN $\frac{E \vdash x : T \quad E \vdash e : T}{E \vdash x = e : T}$ Note: We have a choice about the return value. We could follow ML-style and give the result type 'unit' instead of T .

CIS 341: Compilers5

Arrays

- Array constructs are not hard either
- First: add a new type constructor: $T[]$

NEW $\frac{E \vdash e_1 : \text{int} \quad E \vdash e_2 : T}{E \vdash \text{new } T[e_1](e_2) : T[]}$ e_1 is the size of the newly allocated array. e_2 initializes the elements of the array.

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

UPDATE $\frac{E \vdash e_1 : T[] \quad E \vdash e_2 : \text{int} \quad E \vdash e_3 : T}{E \vdash e_1[e_2] = e_3 : \text{unit}}$ Note: These rules don't ensure that the array index is in bounds – that should be checked dynamically.

CIS 341: Compilers6

Tuples

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

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

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

CIS 341: Compilers 7

References

- ML-style references
- First, add a new type constructor: $T \text{ ref}$

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

Note the similarity with the rules for arrays...

CIS 341: Compilers 8

Recursive Definitions

- Consider the factorial function:


```
int fact(int x) {
  if (x == 0) 1 else x * fact(x-1)
}
```
- Note that the function name **fact** appears inside the body of **fact**'s definition!
- To typecheck the body of fact, we must assume that the type of fact is already known.

$$\frac{E, \text{fact} : \text{int} \rightarrow \text{int}, x : \text{int} \vdash e_{\text{body}} : \text{int}}{E \vdash \text{int fact(int x) (} e_{\text{body}} \text{) : int} \rightarrow \text{int}}$$

- In general: Collect the names and types of all mutually recursive definitions, add them all to the context E before checking any of the definition bodies.

CIS 341: Compilers 9

What are types, anyway?

- A *type* is just a predicate on the set of values in a system.
 - For example, the type "int" can be thought of as a boolean function that returns "true" on integers and "false" otherwise.
 - Equivalently, we can think of a type as just a subset of all values.
- For efficiency and tractability, the predicates are usually taken to be very simple.
- We can easily add new types that distinguish different subsets of values:


```
type tp =
| IntT      (* type of integers *)
| PosT | NegT | ZeroT (* refinements of ints *)
| BoolT    (* type of booleans *)
| TrueT | FalseT    (* subsets of booleans *)
| AnyT     (* any value *)
```

CIS 341: Compilers 10

Modifying the typing rules

- We need to refine the typing rules too...
- Some easy cases:
 - Just split up the integers into their more refined cases:

P-INT

$i > 0$

$E \vdash i : \text{Pos}$

N-INT

$i < 0$

$E \vdash i : \text{Neg}$

ZERO

$E \vdash 0 : \text{Zero}$
- Same for booleans:

TRUE

$E \vdash \text{true} : \text{True}$

FALSE

$E \vdash \text{false} : \text{False}$

CIS 341: Compilers 11

What about "if"?

- Two cases are easy:

IF-T $E \vdash e_1 : \text{True} \quad E \vdash e_2 : T$

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

IF-F $E \vdash e_1 : \text{False} \quad E \vdash e_3 : T$

$E \vdash \text{if } (e_1) e_2 \text{ else } e_3 : T$
- What happens when we don't know statically which branch will be taken?
- Consider the typechecking problem:

$x : \text{bool} \vdash \text{if } (x) 3 \text{ else } -1 : ?$
- The true branch has type Pos and the false branch has type Neg.
 - What should be the result type of the whole if?

CIS 341: Compilers 12

Subtyping and Upper Bounds

- If we think of types as sets of values, we have a natural inclusion relation: $\text{Pos} \subseteq \text{Int}$
- This subset relation gives rise to a *subtype* relation: $\text{Pos} <: \text{Int}$
- Such inclusions give rise to a *subtyping hierarchy*:

```

    graph BT
      Any --> Int
      Any --> Bool
      Int --> Neg
      Int --> Zero
      Int --> Pos
      Bool --> True
      Bool --> False
  
```

- Given any two types T_1 and T_2 , we can calculate their *least upper bound* (LUB) according to the hierarchy.
 - Example: $\text{LUB}(\text{True}, \text{False}) = \text{Bool}$, $\text{LUB}(\text{Int}, \text{Bool}) = \text{Any}$
 - Note: might want to add types for "NonZero", "NonNegative", and "NonPositive" so that set union on values corresponds to taking LUBs on types.

CIS 341: Compilers 13

"If" Typing Rule Revisited

- For statically unknown conditionals, we want the return value to be the LUB of the types of the branches:

$$\frac{\text{IF-BOOL} \quad E \vdash e_1 : \text{bool} \quad E \vdash e_2 : T_1 \quad E \vdash e_3 : T_2}{E \vdash \text{if } (e_1) e_2 \text{ else } e_3 : \text{LUB}(T_1, T_2)}$$

- Note that $\text{LUB}(T_1, T_2)$ is the most precise type (according to the hierarchy) that is able to describe any value that has either type T_1 or type T_2 .
- In math notation, $\text{LUB}(T_1, T_2)$ is sometimes written $T_1 \vee T_2$
- LUB is also called the *join* operation.

CIS 341: Compilers 14

Soundness of Subtyping Relations

- We don't have to treat *every* subset of the integers as a type.
 - e.g., we left out the type NonNeg
- A subtyping relation $T_1 <: T_2$ is *sound* if it approximates the underlying semantic subset relation.
- Formally: write $[[T]]$ for the subset of values of type T
 - i.e. $[[T]] = \{v \mid v : T\}$
 - e.g. $[[\text{Zero}]] = \{0\}$, $[[\text{Pos}]] = \{1, 2, 3, \dots\}$
- If $T_1 <: T_2$ implies $[[T_1]] \subseteq [[T_2]]$, then $T_1 <: T_2$ is sound.
 - e.g. $\text{Pos} <: \text{Int}$ is sound, since $\{1, 2, 3, \dots\} \subseteq \{\dots, -3, -2, -1, 0, 1, 2, 3, \dots\}$
 - e.g. $\text{Int} <: \text{Pos}$ is not sound, since it is *not* the case that $\{\dots, -3, -2, -1, 0, 1, 2, 3, \dots\} \subseteq \{1, 2, 3, \dots\}$

CIS 341: Compilers 15

Soundness of LUBs

- Whenever you have a sound subtyping relation, it follows that: $[[\text{LUB}(T_1, T_2)]] \supseteq [[T_1]] \cup [[T_2]]$
 - Note that the LUB is an overapproximation of the "semantic union"
 - Example: $[[\text{LUB}(\text{Zero}, \text{Pos})]] = [[\text{Int}]] = \{\dots, -3, -2, -1, 0, 1, 2, 3, \dots\} \supseteq \{0, 1, 2, 3, \dots\} = \{0\} \cup \{1, 2, 3, \dots\} = [[\text{Zero}]] \cup [[\text{Pos}]]$
- Using LUBs in the typing rules yields sound approximations of the program behavior (as if the IF-B rule).
- It just so happens that LUBs on types $<: \text{Int}$ correspond to +

$$\frac{\text{ADD} \quad E \vdash e_1 : T_1 \quad E \vdash e_2 : T_2 \quad T_1 <: \text{Int} \quad T_2 <: \text{Int}}{E \vdash e_1 + e_2 : T_1 \vee T_2}$$

CIS 341: Compilers 16

Downcasting

- What happens if we have an Int but need something of type Pos?
 - At compile time, we don't know whether the Int is greater than zero.
 - At run time, we do.
- Add a "checked downcast"

$$\frac{E \vdash e_1 : \text{Int} \quad E, x : \text{Pos} \vdash e_2 : T_2 \quad E \vdash e_3 : T_3}{E \vdash \text{ifPos } (x = e_1) e_2 \text{ else } e_3 : T_2 \vee T_3}$$
- At runtime, ifPos checks whether $e_1 > 0$. If so, branches to e_2 and otherwise branches to e_3 .
- Inside the expression e_2 , x is the name for e_1 's value, which is known to be strictly positive because of the dynamic check.
- Note that such rules force the programmer to add the appropriate checks
 - We could give integer division the type: $\text{Int} \rightarrow \text{NonZero} \rightarrow \text{Int}$

CIS 341: Compilers 17

Extending Subtyping to Other Types

- What about subtyping for tuples?
 - Intuition: whenever a program expects something of type $S_1 * S_2$, it is sound to give it a $T_1 * T_2$.
 - Example: $(\text{Pos} * \text{Neg}) <: (\text{Int} * \text{Int})$

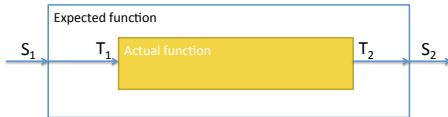
$$\frac{T_1 <: S_1 \quad T_2 <: S_2}{(T_1 * T_2) <: (S_1 * S_2)}$$

- What about functions?
- When is $T_1 \rightarrow T_2 <: S_1 \rightarrow S_2$?

CIS 341: Compilers 18

Subtyping for Function Types

- One way to see it:



- Need to convert an S_1 to a T_1 and T_2 to S_2 , so the argument type is *contravariant* and the output type is *covariant*.

$$\frac{S_1 <: T_1 \quad T_2 <: S_2}{(T_1 \rightarrow T_2) <: (S_1 \rightarrow S_2)}$$

CIS 341: Compilers

19

Subtyping and References

- What is the proper subtyping relationship for references and arrays?

- Suppose we have `NonZero` as a type and the division operation has type: `Int -> NonZero -> Int`

- Recall that `NonZero <: Int`

- Should `(NonZero ref) <: (Int ref)` ?

- Consider this program:

```
Int bad(NonZero ref r) (
  Int ref a = r; (* OK because (NonZero ref <: Int ref *)
  a := 0;        (* OK because 0 : Zero <: Int *)
  42 / !r        (* OK because !r has type NonZero *)
)
```

CIS 341: Compilers

20

Mutable Structures are Invariant

- Covariant reference types are unsound (as demonstrated in the previous example).
- Contravariant reference types are also unsound
 - i.e. If $T_1 <: T_2$ then `ref T2 <: ref T1` is also unsound
 - Exercise: construct a program that breaks contravariant references.
- Moral: Mutable structures are invariant: `T ref <: T ref`
- Same holds for arrays, OCaml-style mutable records, object fields, etc.
 - Note: Java gets this wrong. It allows covariant array subtyping, but then it compensates by adding a dynamic check on every array update!

CIS 341: Compilers

21