Announcements / Plan

• HW5: OAT – typechecking, structs, function pointers
  – Available soon
  – Due: Thursday, April 13

• HW6: LLVM Optimization: analysis and register allocation
  – Due: Wednesday, April 26

• FINAL EXAM: Thursday, May 4th noon – 2:00p.m.
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…
Beyond describing “structure”… describing “properties”
Types as sets
Subsumption

TYPES, MORE GENERALLY
Tuples

- ML-style tuples with statically known number of products:
- First: add a new type constructor: \( T_1 \ast \cdots \ast T_n \)

\[
\begin{align*}
& \text{TUPLE} \\
& E \vdash e_1 : T_1 \quad \cdots \quad E \vdash e_n : T_n \\
& \hline
& E \vdash (e_1, \ldots, e_n) : T_1 \ast \cdots \ast T_n \\
& \text{PROJ} \\
& E \vdash e : T_1 \ast \cdots \ast T_n \quad 1 \leq i \leq n \\
& \hline
& E \vdash \#i \ e : T_i
\end{align*}
\]
ML-style references (note that ML uses only expressions)

First, add a new type constructor: $\text{T ref}$

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

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

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

Note the similarity with the rules for arrays…
Arrays

- Array constructs are not hard either, here is one possibility
  - First: add a new type constructor: \( T[] \)

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

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

Note: These rules don’t ensure that the array index is in bounds – that should be checked dynamically.
• What is the type of null?
• Consider:
  
  ```java
  int[] a = null;   // OK?
  int x   = null;   // not OK?
  string s = null; // OK?
  ```

  $E \vdash \text{null} : r$

• Null has any *reference type*
  
  – Null is generic

• What about type safety?
  
  – Requires defined behavior when dereferencing null
e.g. Java's `NullPointerException`
  
  – Requires a safety check for every dereference operation
    (typically implemented using low-level hardware "trap" mechanisms.)
Recursive Definitions

- Consider the ML factorial function:
  
  ```ml
  let rec fact (x:int) : int =
      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.

  ```
  E, fact : int -> int, x : int ⊢ e_{body} : int
  ```

  ```
  E ⊢ int fact(int x) ( e_{body}) : int -> 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.

- Often useful to separate the “global context” from the “local context”
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.
  – Types are an *abstraction* mechanism

• We can easily add new types that distinguish different subsets of values:

```plaintext
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
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:

\[
\begin{align*}
\text{P-INT} & \quad \text{N-INT} & \quad \text{ZERO} \\
\begin{array}{c}
i > 0 \\
E \vdash i : \text{Pos}
\end{array} & \quad \begin{array}{c}
i < 0 \\
E \vdash i : \text{Neg}
\end{array} & \quad \begin{array}{c}
E \vdash 0 : \text{Zero}
\end{array}
\end{align*}
\]

• Same for booleans:

\[
\begin{align*}
\text{TRUE} & \quad \text{FALSE} \\
E \vdash \text{true} : \text{True} & \quad E \vdash \text{false} : \text{False}
\end{align*}
\]
What about “if”? 

- Two cases are easy:

\[
\begin{align*}
\text{IF-T} & \quad E \vdash e_1 : \text{True} \quad E \vdash e_2 : T \\
\text{IF-F} & \quad E \vdash e_1 : \text{False} \quad E \vdash e_3 : T
\end{align*}
\]

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

- What happens when we don’t know statically which branch will be taken?
- Consider the typechecking problem:

\[
x: \text{bool} \vdash \text{if (} x \text{) } 3 \text{ else } -1 : ?
\]

- The true branch has type Pos and the false branch has type Neg.
  - What should be the result type of the whole if?
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:

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.
“If” Typing Rule Revisited

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

\[
\frac{E \vdash e_1 : \text{bool} \quad E \vdash e_2 : T_1 \quad E \vdash e_3 : T_2}{E \vdash \text{if} (e_1) \ e_2 \ \text{else} \ e_3 : \text{LUB}(T_1, T_2)}
\]

• 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 \( T_1 \lor T_2 \)
• LUB is also called the \( \text{join} \) operation.
Subtyping Hierarchy

- A *subtyping hierarchy*:

  ![Subtyping Hierarchy Diagram]

  - Reflexive: $T <: T$ for any type $T$
  - Transitive: $T_1 <: T_2$ and $T_2 <: T_3$ then $T_1 <: T_3$
  - Antisymmetric: If $T_1 <: T_2$ and $T_2 <: T_1$ then $T_1 = T_2$

---

CIS 341: Compilers
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 (closed) values of type $T$
  - i.e. $⟦T⟧ = \{v \mid ⊢ v : T\}$
  - e.g. $⟦\text{Zero}⟧ = \{0\}$, $⟦\text{Pos}⟧ = \{1, 2, 3, \ldots\}$

- If $T_1 <: T_2$ implies $⟦T_1⟧ \subseteq ⟦T_2⟧$, then $T_1 <: T_2$ is sound.
  - e.g. Pos $<: \text{Int}$ is sound, since $\{1,2,3,\ldots\} \subseteq \{\ldots,-3,-2,-1,0,1,2,3,\ldots\}$
  - e.g. Int $<: \text{Pos}$ is not sound, since it is not the case that $\{\ldots,-3,-2,-1,0,1,2,3,\ldots\} \subseteq \{1,2,3,\ldots\}$
Soundness of LUBs

• Whenever you have a sound subtyping relation, it follows that:
  \[ \llbracket \text{LUB}(T_1, T_2) \rrbracket \supseteq \llbracket T_1 \rrbracket \cup \llbracket T_2 \rrbracket \]
  
  – Note that the LUB is an over approximation of the “semantic union”
  
  – Example: \[ \llbracket \text{LUB}(\text{Zero}, \text{Pos}) \rrbracket = \llbracket \text{Int} \rrbracket = \{\ldots, -3, -2, -1, 0, 1, 2, 3, \ldots\} \supseteq \{0, 1, 2, 3, \ldots\} = \llbracket \text{Zero} \rrbracket \cup \llbracket \text{Pos} \rrbracket \]

• 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 <: Int correspond to +

\[
E \vdash e_1 : T_1 \quad E \vdash e_2 : T_2 \quad T_1 <: \text{Int} \quad T_2 <: \text{Int} \\
E \vdash e_1 + e_2 : T_1 \lor T_2
\]
Subsumption Rule

• When we add subtyping judgments of the form \( T <: S \) we can uniformly integrate it into the type system generically:

\[
\text{SUBSUMPTION} \quad E \vdash e : T \quad T <: S
\]

\[
E \vdash e : S
\]

• Subsumption allows any value of type \( T \) to be treated as an \( S \) whenever \( T <: S \).

• Adding this rule makes the search for typing derivations more difficult – this rule can be applied anywhere, since \( T <: T \).
  – But careful engineering of the typing system can incorporate the subsumption rule into a deterministic algorithm.
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”

\[
E \vdash e_1 : \text{Int} \quad E, x : \text{Pos} \vdash e_2 : T_2 \quad E \vdash e_3 : T_3
\]

\[
E \vdash \text{ifPos} (x = e_1) \ e_2 \ \text{else} \ e_3 : T_2 \lor T_3
\]

• At runtime, ifPos checks whether \( e_1 \) is \( > 0 \). 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: \( \text{Int} \to \text{NonZero} \to \text{Int} \)