Announcements

• HW4 Oat v1 out
  • Due Tuesday Oct 29 (7 days)

• Reference solns
  • Will be released on Canvas
  • HW2 later today
  • HW3 later this week
Today

• Types as sets of values
• Subtyping
  • Subsumption
  • Downcasting
• Functions
• Records
• References
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:

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

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

• Same for booleans:

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

- Two cases are easy:

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

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

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

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

```
  Any
  \( \leftarrow \)  \( \rightarrow \)
   \( \downarrow \)  \( \downarrow \)
  Int  \( <\)  \( >\)
  \( \downarrow \)  \( \rightarrow \)  \( \leftarrow \)
 Neg  Zero  Pos  \( <\)  \( =\)  \( >\)
  \( \downarrow \)  \( \rightarrow \)  \( \leftarrow \)
  \( \leftarrow \)  \( =\)  \( \rightarrow \)
 True  False
```

• Given any two types \( T_1 \) and \( T_2 \), we can calculate their **least upper bound** (LUB) according to the hierarchy.
  • Example: \( \text{LUB(True, False)} = \text{Bool} \), \( \text{LUB(Int, Bool)} = \text{Any} \)
  • 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:

  \[
  \text{IF-BOOL} \quad E \vdash e_1 : \text{bool} \quad E \vdash e_2 : T_1 \quad E \vdash e_3 : T_2
  \]

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

- Note: LUB(T1, T2) is the most precise type (according to the hierarchy) that describes any value with either type T1 or type T2.
- Math notation: LUB(T1, T2) is sometimes written T1 ∨ T2 or T1 ⊔ T2
- LUB is also called the join operation.
Subtyping Hierarchy

• A subtyping hierarchy:

- The subtyping relation is a partial order:
  - Reflexive: $T <: T$ for any type $T$
  - Transitive: $T_1 <: T_2$ and $T_2 <: T_3$ then $T_1 <: T_3$
  - Antisymmetric: $T_1 <: T_2$ and $T_2 <: T_1$ then $T_1 = T_2$

![Diagram showing a subtyping hierarchy with types Any, Int, Neg, Zero, Pos, Bool, True, False, and the relationships between them.](image-url)
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 <: Int is sound, since $\{1, 2, 3, \ldots\} \subseteq \{\ldots, -3, -2, -1, 0, 1, 2, 3, \ldots\}$
  - e.g., Int <: 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:
  \[
  \llbracket T_1 \rrbracket \cup \llbracket T_2 \rrbracket \subseteq \llbracket \text{LUB}(T_1, T_2) \rrbracket
  \]
- Note that the LUB is an over approximation of the “semantic union”
- Example: \[
  \llbracket \text{Zero} \rrbracket \cup \llbracket \text{Pos} \rrbracket = \{0\} \cup \{1,2,3,\ldots\}
  = \{0,1,2,3,\ldots\}
  \subseteq \{\ldots,-3,-2,-1,0,1,2,3,\ldots\}
  = \llbracket \text{Int} \rrbracket = \llbracket \text{LUB}(\text{Zero, Pos}) \rrbracket
  \]
- Using LUBs in the typing rules yields sound approximations of the program behavior (as in the IF-BOOL rule).

\[
\begin{align*}
\text{IF-BOOL} \\
E \vdash e_1 : \text{bool} & \quad E \vdash e_2 : T_1 & \quad E \vdash e_3 : T_2 \\
\hline
E \vdash \text{if } (e_1) \ e_2 \ \text{else } e_3 : T_1 \lor T_2
\end{align*}
\]
Subsumption Rule

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

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

• **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”

\[
\begin{align*}
E &\vdash \text{ifPos } (x = e_1) \ e_2 \ \text{else } \ e_3 : T_2 \lor T_3 \\
E &\vdash e_1 : \text{Int} \quad E, \ x : \text{Pos} \vdash e_2 : T_2 \quad E \vdash e_3 : T_3
\end{align*}
\]

• At runtime, ifPos checks whether \( e_1 \) is > 0. If so, branches to \( e_2 \) and otherwise branches to \( e_3 \)
• Inside expression \( e_2 \), \( x \) is \( e_1 \)’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: \( \text{Int -> NonZero -> Int} \)
Extending Subtyping to Other Types

- What about subtyping for tuples?
  - When a program expects a value of type $S_1 \times S_2$, when is sound to give it a $T_1 \times T_2$?

  $$T_1 <: S_1 \quad T_2 <: S_2$$

  $$(T_1 \times T_2) <: (S_1 \times S_2)$$

- Example: $(\text{Pos} \times \text{Neg}) <: (\text{Int} \times \text{Int})$

- What about functions?
  - When is $T_1 \rightarrow T_2 <: S_1 \rightarrow S_2$ ?
  - When a program expects a function of type $S_1 \rightarrow S_2$, when can we give it a function of type $T_1 \rightarrow T_2$ ?
Subtyping for Function Types

• One way to see it:

\[ S_1 <: T_1 \quad T_2 <: S_2 \]
\[ (T_1 \rightarrow T_2) <: (S_1 \rightarrow S_2) \]

• Need to convert an \( S_1 \) to a \( T_1 \) and \( T_2 \) to \( S_2 \), so the argument type is \textit{contravariant} and the output type is \textit{covariant}. 
Immutable Records

• Record type: \{lab_1:T_1; lab_2:T_2; \ldots ; lab_n:T_n\}
  • Each lab_i is a label drawn from a set of identifiers.

\[
\begin{align*}
E \vdash e_1 : T_1 \\
E \vdash e_2 : T_2 \\
\vdots \\
E \vdash e_n : T_n \\
\overline{\quad} \\
E \vdash \{lab_1 = e_1; lab_2 = e_2; \ldots ; lab_n = e_n\} : \{lab_1:T_1; lab_2:T_2; \ldots ; lab_n:T_n\}
\end{align*}
\]

\[
\text{RECORD}
\]

\[
\begin{align*}
E \vdash e : \{lab_1:T_1; lab_2:T_2; \ldots ; lab_n:T_n\} \\
\overline{\quad} \\
E \vdash e.lab_i : T_i
\end{align*}
\]

\[
\text{PROJECTION}
\]
Immutable Record Subtyping

• Depth subtyping:
  • Corresponding fields may be subtypes

\[
\begin{align*}
T_1 & <: U_1 & T_2 & <: U_2 & \ldots & T_n & <: U_n \\
\{lab_1:T_1; lab_2:T_2; \ldots ; lab_n:T_n\} & <: & \{lab_1:U_1; lab_2:U_2; \ldots ; lab_n:U_n\}
\end{align*}
\]

• Width subtyping:
  • Subtype record may have more fields:

\[
\begin{align*}
\text{WIDTH} & & m \leq n \\
\{lab_1:T_1; lab_2:T_2; \ldots ; lab_n:T_n\} & <: & \{lab_1:T_1; lab_2:T_2; \ldots ; lab_m:T_m\}
\end{align*}
\]
Depth & Width Subtyping vs. Layout

- Width subtyping (without depth) is compatible with “inlined” record representation as with C structs:

  ```
  x y z
  {x:int; y:int; z:int} <: {x:int; y:int}
  ```

  The layout and underlying field indices for `x` and `y` are identical.
  The `z` field is just ignored.

- Depth subtyping (without width) is similarly compatible, assuming that the space used by A is the same as the space used by B whenever `A <: B`

- But… they don't mix. Why?
Immutable Record Subtyping (cont’d)

• Width subtyping assumes an implementation in which order of fields in a record matters:
  \[ \{x: \text{int}; y: \text{int}\} \neq \{y: \text{int}; x: \text{int}\} \]
But: \( \{x: \text{int}; y: \text{int}; z: \text{int}\} <: \{x: \text{int}; y: \text{int}\} \)
  • Implementation: a record is a struct, subtypes just add fields at the end of the struct.

• Alternative: allow permutation of record fields:
  \[ \{x: \text{int}; y: \text{int}\} = \{y: \text{int}; x: \text{int}\} \]
  • Implementation: compiler sorts the fields before code generation.
  • Need to know all of the fields to generate the code

• Permutation is not directly compatible with width subtyping:
  \[ \{x: \text{int}; z: \text{int}; y: \text{int}\} = \{x: \text{int}; y: \text{int}; z: \text{int}\} \neq: \{y: \text{int}; z: \text{int}\} \]
If you want both:

- If you want permutability & dropping, you need to either copy (to rearrange the fields) or use a dictionary like this:

```
p = {x=42; y=55; z=66}:{x:int; y:int; z:int}
q : {y:int; z:int} = p
```
Mutability and Subtyping

• What about when we add mutable locations?
  • References, arrays, ...
• What is the type of `null`?

• Consider:
  - `int[ ] a = null;`  // OK?
  - `int x = null;`  // not OK?
  - `string s = null;`  // OK?

• 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.)
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: \( \text{Int} \rightarrow \text{NonZero} \rightarrow \text{Int} \)
  • Recall that \( \text{NonZero} <: \text{Int} \)

• Should \((\text{NonZero ref}) <: (\text{Int ref})\) ?

• Consider this program:

```c
Int bad(NonZero ref r) {
    Int ref a = r;   (* OK because (NonZero ref <: Int ref*)
    a := 0;          (* OK because 0 : Zero <: Int *)
    return (42 / !r) (* OK because !r has type NonZero *)
}
```
Mutable Structures are Invariant

• Covariant reference types are unsound
  • As demonstrated in the previous example

• Contravariant reference types are also unsound
  • i.e. If T1 <: T2 then ref T2 <: ref T1 is also unsound
  • Exercise: construct a program that breaks contravariant references.

• Moral: Mutable structures are invariant:
  T1 ref <: T2 ref implies T1 = T2

• Same holds for arrays, mutable records, object fields, etc.
  • Note: Java and C# get this wrong. They allow covariant array subtyping, but then compensate by adding a dynamic check on every array update!
Another Way to See It

• We can think of a reference cell as an immutable record (object) with two functions (methods) and some hidden state:

\[ T \text{ ref } \approx \{ \text{get: } \text{unit } \to T; \text{ set: } T \to \text{unit} \} \]

• get returns the value hidden in the state.
• set updates the value hidden in the state.

• When is \( T \text{ ref } <: S \text{ ref} \)?

• Records are like tuples: subtyping extends pointwise over each component.

\{ \text{get: } \text{unit } \to T; \text{ set: } T \to \text{unit} \} <: \{ \text{get: } \text{unit } \to S; \text{ set: } S \to \text{unit} \}

• get components are subtypes: \( \text{unit } \to T \ <: \text{unit } \to S \)
• set components are subtypes: \( T \to \text{unit } \ <: S \to \text{unit} \)

• From get, we must have \( T <: S \) (covariant return)
• From set, we must have \( S <: T \) (contravariant arg.)
• From \( T <: S \) and \( S <: T \) we conclude \( T = S \).
Structural vs. Nominal Typing

- Is type equality / subsumption defined by the **structure** of the data or the **name** of the data?
- Example 1: type abbreviations (OCaml) vs. “newtypes” (a la Haskell)

```ocaml
(* OCaml: *)
type cents = int  (* cents = int in this scope *)
type age = int

let foo (x:cents) (y:age) = x + y
```

```haskell
(* Haskell: *)
newtype Cents = Cents Integer  (* Integer and Cents are isomorphic, not identical. *)
newtype Age = Age Integer

foo :: Cents -> Age -> Int
foo x y = x + y                (* Ill typed! *)
```

- Type abbreviations are treated “structurally”
- Newtypes are treated “by name”
Nominal Subtyping in Java

• In Java, Classes and Interfaces must be named and their relationships explicitly declared.

```java
/* Java: */
interface Foo {
    int foo();
}

class C {
    /* Does not implement the Foo interface */
    int foo() {return 2;}
}

class D implements Foo {
    int foo() {return 42;}
}
```

• Similarly for inheritance: programmers must declare the subclass relation via the “extends” keyword.
  • Typechecker still checks that the classes are structurally compatible.