Lecture 20: Subtyping & OO Languages

CSC 131
Spring, 2019
Kim Bruce

Subtyping

- Can be added to non-OO languages
- Matching structures and signatures similar
  - but more restricted.
- Provides support for using values from new types in old (unexpected) contexts

Subtype Polymorphism

S is a subtype of T, written S <: T,

iff

a value of type S can be used in any context
expecting a value of type T,

i.e., S can masquerade as a T.

Subsumption: e: S & S <: T ⇒ e: T.

Immutable Records

Records without field update (like Haskell/ML):
Sandwich = { bread: BreadType;
             filling: FoodType }
s: Sandwich = { bread = rye;
               filling = pastrami }
Only operation is extracting field:
... s.filling ...

...
Specializing Record Types

CheeseSandwich = {bread: BreadType;
filling: CheeseType;
sauce: SauceType}
c_s: CheeseSandwich = {bread = white;
filling = cheddar;
sauce = mustard}

Subtyping Immutable Records

If $r:\{l_i:T_i\}_{i \leq k}$ then expect $r.l_i:T_i$.

When is $\{l_i:T'_i\}_{i \leq n} <:\{l_i:T_i\}_{i \leq k}$?

Suppose $r':\{l_i:T'_i\}_{i \leq n}$

When can $r'$ masquerade as elt of $\{l_i:T_i\}_{i \leq k}$?

Need $r'.l_i:T_i$.

Masquerading

\[
\begin{array}{c|c|c|c|c}
\hline
l_1 & l_2 & l_3 & l_4 \\hline
\end{array}
\]

\[
\begin{array}{c|c|c|c|c}
\hline
r_1 & r_2 & r_3 & r_4 \\hline
\end{array}
\]

\[
\{l_i:T_i\}_{i \leq n} <:\{l_i:T_i\}_{i \leq k}
\]

iff

$k \leq n$ and for all $1 \leq i \leq k$, $T'_i <: T_i$.

Functions

If $f:S \rightarrow T$ and $s:S$ then $f(s):T$.

When is $S' \rightarrow T' <: S \rightarrow T$?

If $f':S' \rightarrow T'$, need $f'(s):T$.
Subtyping Functions

\[ S \rightarrow_{S'} \rightarrow_{T'} \rightarrow_{T} \]

\[ S' \rightarrow T' \ll S \rightarrow T \]

iff

\[ S \ll S' \text{ and } T' \ll T. \]

Contravariant for parameter types.
Covariant for result types.

Variables

Variables can be suppliers & receivers of values.

\[ x := x + 1 \]

If \( x \) is a variable of type \( T \), write \( x: \text{ref } T \).

When is \( \text{ref } T' \ll \text{ref } T \)?

To replace variable \( x: \text{ref } T \) by \( x': \text{ref } T' \) in:

- expression: \( \ldots x \ldots \)

Need \( T' \ll T \).
- assignment: \( x := e \) where \( e: T \).

Need \( T \ll T' \).

Variables

\[ x := T \]

Supplier: covariant;
Receiver: contravariant

\[ \text{ref } T' \ll \text{ref } T \iff T' \approx T \]

Exercises

Updatable Records:

When is \( \{ l_i : T'_i \}_{i=six} \ll \{ l_i : T_i \}_{i=six} \)?

\[ \ldots r.l_i := e \ldots \]
More Exercises

Arrays:
• If $S <: T$, is Array of $S <$: Array of $T$?
Java says yes, but ...

not safe!

With few exceptions, for $F$: Types $\rightarrow$ Types,
$S <: T \nRightarrow F(S) <: F(T)$.

Object-Oriented Languages

Roots in ADT Languages

• Ada and Modula-2 internal reps
  - couldn't be instantiated dynamically
  - no type or other method of organizing, despite similarity to records
  - provide better modules for building large systems
• Called object-based

Responding to the  “SOFTWARE CRISIS!”
Qualities Desired in Software

- Correctness
- Robustness
- Extensibility *Almost all supported by ADT's*
- Reusability
- Compatibility

Object-Oriented Languages

- Objects that are data abstractions
- Objects have associated object type (classes or interfaces)
- Classes may *inherit* attributes from superclass
- Computations proceed by sending messages
- Subtype polymorphism
- Support dynamic method invocation

Programming Objects in ML

```ml
exception Empty;
fun newStack(x) = 
  let
    val store = ref [x]
  in
    {push = fn y => store := y:(!store); 
pop = fn z => case !store of 
              Nil = raise Empty
              | (y::ys) = (!store := ys; y)
    }
  end;
val myStack = newStack(0);
#push(myStack)(1);
#pop(myStack)(0);
```

Weakness of ML

- No subtyping
- No this/self
- No inheritance
- *Similar issues in trying to do objects in LISP or other functional languages.*
- *Haskell doesn't have state!*
**OO Keywords**

- Object
- Message
- Class
- Instance
- Method
- Subtype
- Subclass

**Objects**

- Internal data abstractions
- Hide representation
- Have associated state
- Methods have access to its state
- Self

**Object Types**

- Allow objects to be first class
- Allow use in assignment, parameters, components of structures
- Allow objects to be classified via subtyping