CS410/510
Compiling Advanced Languages
Spr2000 Lecture Notes
Subtyping II

Minimal Typings

If we add subsumption law to our type system, the resulting system $\triangleright_S$ is no longer **syntax-directed** (or **algorithmic**), because subsumption can be applied at many places.

Thus, it is not clear how to write a **type-checking algorithm** or even whether every term has a unique type at all (in fact, in $\triangleright_S$ it doesn't!).

Fortunately, we can formulate a revised type system $\triangleright_{S'}$ that types the same terms as $\triangleright_S$, is syntax-directed, and gives unique **minimal** types to terms.

I.e., if $E \triangleright_S M : \tau$, then

$E \triangleright_{S'} M : \tau'$for some$\tau' <: \tau$

The basic idea behind $\triangleright_{S'}$ is to delay uses of subsumption to the latest possible point in a derivation. Under this policy, it turns out that subsumption is only needed in two rules:

$$\frac{E \triangleright M : \tau_1 \rightarrow \tau_2 \quad E \triangleright N : \tau_1' \quad \tau_1' <: \tau_1}{E \triangleright (MN) : \tau_2} \text{S-APP}$$

$$\frac{M :< l_1 : \tau_1, \ldots, l_n : \tau_n > \quad E \triangleright M_i : \tau_i \rightarrow \mu_i \quad \mu = \mu_1 \sqcup \ldots \sqcup \mu_n \quad (l_i \textit{ distinct})}{E \triangleright \texttt{switch } M \texttt{ of } l_1 \texttt{ => } M_1 \texttt{ | } \ldots \texttt{ | } l_n \texttt{ => } M_n : \mu} \text{(S-SWITCH)}$$

Joins and Meets

The latter rule makes use of the **join** operator ($\sqcup$), which finds the smallest common supertype of two types. That is:

$\tau = \tau_1 \sqcup \tau_2$ iff

- $\tau_1 <: \tau$
- $\tau_2 <: \tau$
- for any type μ, if $\tau_1 <: \mu$ and $\tau_2 <: \mu$, then $\tau <: \mu$.

Note that joins don't always exist; for example, a record type and a sum type have no join. But in the subtyping partial order, any two types that have a common supertype have a (unique) join.

There is a symmetric **meet** operator ($\sqcap$), which finds the largest common subtype of two types. That is:

$\tau = \tau_1 \sqcap \tau_2$ iff

- $\tau <: \tau_1$
- $\tau <: \tau_2$
- for any type μ, if $\mu <: \tau_1$ and $\mu <: \tau_2$, then $\mu <: \tau$.

Again, meets exist for any two types that have a common subtype at all.

Calculating joins and meets

Calculating joins is just a little tricky:

- For records, we take the **intersection** of the label sets and then the joins of corresponding field types.

- For sums, we take the **union** of the label sets and, for common labels, the joins of corresponding field types.

- For arrows, we take the join of the result types and the **meet** of the argument types. (Covariance at work again!)

Meets are just the same, only backwards! (Exercise: what does "backwards" mean?)

Modeling Objects

The "traditional and obvious" way to model objects is to treat each object as a record, whose components are the fields and methods of the object.

```
class C {
  int x;
  int y;
  C(int x0,y0) { x = x0; y = y0; }
  int f(int t) { return x + t; }
  int g(int t) { return y + f(t); }
}
C w = new C (1,12);
```

The most naive layout:

```
          ------------
w ----> | f = Code1|
          ------------
          | g = Code2|
          ------------
          |  x = 1   |
          ------------
          |  y = 12  |
          ------------
```

Less naive More commonly, we store methods indirectly in a separate record shared by all class instances:

```
C v = new C(2,24);
                                           Cmethods
           ------------               -------------
w ----> | methods =|------------>|  f = Code1|
           ------------      |        -------------
           |  x = 1    |      |        |  g = Code2|
           ------------      |        -------------
           |  y = 12   |      |
           ------------      |
                             |
           ------------      |
v ----> | methods =|-----|
           ------------
           |  x = 2    |
           ------------
           |  y = 24   |
           ------------
```

Modeling Inheritance

Introducing subclasses:

```
class C1 extends C {
   int z;
   C1 (int x0,y0,z0) { super(x0,y0); z = z0; }
   int f(t:int) { return z + t; }
   int h(t:int) { return x - t; }
}
C1 w1 = new C1(3,36,99);
```

Objects of subclass get **all** fields of superclass plus any new ones. Method records can be linked, requiring a search of super-classes.

```
         ------------            -------------
w ----> | methods =|----------->|  f = Code1|
         ------------     |      -------------
        |  x = 1   |      |     |  g = Code2|
         ------------     |      -------------
        |  y = 12  |      |
         ------------     |------------|
                                       |
           ----------            ------|------
w1 ---> | methods =|----------->|     ^     |
         ------------            -------------
        |  x = 3   |            |  h = Code3|
         ------------            -------------
        |  y = 36  |            |  f = Code4|
         ------------            -------------
        |  z = 99  |
         ------------
```

Flattened method records

More commonly, method records for subclasses can be flattened, by duplicating entries from superclasses:

```
                                          Cmethods
              ------------                -------------
   w ----> | methods =|---------->|  f = Code1|
              ------------                -------------
              |  x = 1   |                |  g = Code2|
              ------------                -------------
              |  y = 12  |
              ------------
                                          C1methods
              ------------                -------------
   w1 ---> | methods =|---------->|  f = Code4|
              ------------                -------------
              |  x = 3   |                |  g = Code2|
              ------------                -------------
              |  y = 36  |                |  h = Code3|
              ------------                -------------
              |  z = 99  |
              ------------
```

Note that these records obey the usual subsumption requirements, so we can use a `C1` object wherever a `C` object is expected.

In particular, as long as we stick to **single** inheritance, subclass representations are just **extensions** of superclass ones, so we can use strictly **ordered** records and refer to the fields by **offsets**. For example, field `y` is always at offset 2 in **any** object of class `C` or any of its subclasses. (The same is true for methods in the flattened representation.)

Encoding Method Invocation

We can encode objects into our **untyped** IR in a simple way: records are records; methods are closed functions.

But note that a method body has access to the fields of the class for which it was defined. To handle this, the IR version of each method function is passed an extra initial argument that points to the object itself.

So a typical method invocation like

```
w.f(10);
```

is encoded by a combination of record selection and self-application:

```
w.methods.f(w,10);
```

Encoding Method Definitions

Here are the translations of the methods:

```
C:  int f(int t) { return x + t; }
C:  int g(int t) { return y + f(t); }
C1: int f(int t) { return z + t; }
C2: int h(int t) { return x - t; }

Cmethods =
{f = λ(self,t) -> self.x + t,
g = λ(self,t) ->
          self.y + self.methods.f(self,t)}

C1methods =
{f = λ(self,t) -> self.z + t,
g = Cmethods.g,
h = λ(self,t) -> self.x - t;}
```

(These augmented functions are sometimes called **pre-methods**.)

The object constructor mechanism builds records accessing the appropriate methods:

```
C w = new C (1,12);
C1 w1 = new C1(3,36,99);

w = {methods = Cmethods, x = 1, y = 12}
w1 = {methods = C1methods, x = 3, y = 36, z = 99}
```

Dynamic binding

Note that methods defined in a superclass can end up executing code defined in an overriding method of a subclass! E.g.,

```
w.g(0)  --->
  w.methods.g(w,0) =
  Cmethods.g(w,0) =
  w.y + w.methods.f(w,0)
  w.y + Cmethods.f(w,0) =
  w.y + w.x + 0 =
  13

w1.g(0) --->
  w1.methods.g(w1,0) =
  C1methods.g(w1,0) =
  Cmethods.g(w1,0) =
  w1.y + w1.methods.f(w1,0) =
  w1.y + C1methods.f(w1,0) =
  w1.y + w1.z + 0 =
  135
```

Key point: methods defined in a particular class are always invoked with a `self` parameter from that class or one of its subclasses. So a method never attempts to access an undefined field from `self`.

Typing the Encoding

Since we believe that the IR terms generated by this translation evaluate safely, we would hope to be able to type them! But unfortunately, our current type system can't do it in a natural way.

The type of IR records representing objects of class C and C1 are (cheating a little about pairs):

```
C = {methods : {f: (C,Int) -> Int,
                g: (C,Int) -> Int },
     x: Int, y: Int}

C1 = {methods : {f: (C1,Int) -> Int,
                 g: (C1,Int) -> Int,
                 h: (C1,Int) -> Int },
      x: Int, y: Int, z: Int }
```

Leaving aside the problem of recursion (our definition of C refers to C!), which turns out to be easy to fix by extending our system with recursive types, there's a bigger problem: C1 is **not** a subtype of C !

The problem is with the types of the `self` parameters to `f` and `g`: by covariance, the type of (say) `C1.methods.f` is a subtype of `C.methods.f` only if `C` is a subtype of `C1` !

In essence, the type system is too primitive to capture our intended (safe) use of these records, while ruling out other (unsafe) uses.