Object-Oriented Types

sk and dbtucker

October 30, 2000

Casts and subtypes

For today, let's assume we're back in a monomorphic world. Recall the nlist datatype:

```
(datatype nlist
  [nempty]
  [ncons (f num)
         (r nlist)])
```

In the assignment Design 1, we introduced three new types for this datatype: nlist, nemptyT, and nconsT. The constructors now return values of the appropriate variant type:

$$\begin{array}{c} \mathit{nempty} : () \rightarrow \mathsf{nemptyT} \\ \mathit{ncons} : \mathsf{num} \times \mathsf{nlist} \rightarrow \mathsf{nconsT} \end{array}$$

The selectors can only operate on values of the correct variant type:

$$\begin{array}{c} f : \mathsf{nconsT} \rightarrow \mathsf{num} \\ r : \mathsf{nconsT} \rightarrow \mathsf{nlist} \end{array}$$

Let's see if we can define the function *length* using these new types:

```
(define length
  (λ (L : ???)
    (if (nempty? L)
        0
        (+ 1 (length (r L))))))
```

The argument L can be either an nemptyT or an nconsT, so what do we write for the type of the argument? The only type that makes sense is nlist:

```
(define length
  (λ (L : nlist)
    (if (nempty? L)
        0
        (+ 1 (length (r L))))))
```

We want to say that both $\mathsf{nemptyT}$ and nconsT are nlists. How do we write type judgments that express this?

$$\frac{\Gamma \vdash e : \mathsf{nemptyT}}{\Gamma \vdash e : \mathsf{nlist}}$$

$$\frac{\Gamma \vdash e : \mathsf{nconsT}}{\Gamma \vdash e : \mathsf{nlist}}$$

To see how these rules are used, consider the proof tree for $(\mathit{ncons}\ 1\ (\mathit{nempty}))$:

$$\frac{\{\} \vdash 1 : \mathsf{num} \qquad \dfrac{\{\} \vdash (\mathit{nempty}) : \mathsf{nemptyT}}{\{\} \vdash (\mathit{nempty}) : \mathsf{nlist}}\ (\leftarrow \text{ new rule})}{\{\} \vdash (\mathit{ncons}\ 1\ (\mathit{nempty})) : \mathsf{nconsT}}$$

Since *nempty* creates a value of type $\mathsf{nemptyT}$, and the second argument to *ncons* must be an nlist, we need to use the new type judgment.

Are these new rules sufficient for type checking *length*? Let's look at the subexpression $(r\ L)$. The selector r has type:

$$r : \mathsf{nconsT} \rightarrow \mathsf{nlist}$$

There's a problem: r consumes an nconsT, but L is an nlist. We need to say something like:

$(r\ (\mathbf{cast}\ L\ \mathsf{nconsT}))$

The expression $(\mathbf{cast}\ e\ \tau)$ has no run-time semantics—its value is just the value of e. The importance of the **cast** construct is that it tells the type checker to treat expression e as having type τ. The type judgment is thus:

$$\frac{\Gamma \vdash e : \tau'}{\Gamma \vdash (\mathbf{cast}\ e\ \tau) : \tau}$$

How can we tell if **cast** will succeed? Let's look at several cases:

τ	τ'	what to do
nconsT	nlist	upcast—always succeeds
$\mathsf{nemptyT}$	nlist	upcast—always succeeds
nlist	nconsT	downcast—need to check at run-time
nlist	$\mathsf{nemptyT}$	downcast—need to check at run-time
nlist	num	random cast—can reject during type-checking
$\mathsf{num} \rightarrow \mathsf{bool}$	num	random cast—can reject during type-checking

Upcasts always succeed, but downcasts can possibly fail—for example, $(\mathbf{cast}\ (\mathit{nempty})\ \mathsf{nconsT})$ should fail, because otherwise you could call *rest* on an empty list. No matter how you try to implement the *length* function, you need a downcast somewhere. This downcast must be checked at run time to ensure type soundness.

We can state the rule governing casts differently by adding the following restriction:

$$\tau \text{ is above } \tau' \quad \text{or} \quad \tau' \text{ is above } \tau$$

By "above", we mean that the second type is a *subtype* of the first; for example, nemptyT is a subtype of nlist. We denote this by nemptyT <: nlist. So, we can write the restriction as:

$$\tau <: \tau' \quad \text{or} \quad \tau' <: \tau$$

What this means is that the rule only applies when τ and τ' are so related. All other casts are what we call "random casts", and should be rejected immediately by the type checker.

Cases

We want to abstract over this casting so that instead of cluttering our code with **cast** expressions, some other function does the work for us. We can write a function *nlist-cases* that will handle the casting. Its type is:

$$\textit{nlist-cases} : \mathsf{nlist} \times (\mathsf{nemptyT} \to \alpha) \times (\mathsf{nconsT} \to \alpha) \longrightarrow \alpha$$

The function definition is:

```
(define (nlist-cases L : nlist   f1 : (nemptyT → α)   f2 : (nconsT → α))
  (if (nempty? L)
      (f1 (cast L nemptyT))
      (f2 (cast L nconsT))))
```

Now we can write the length function without using casts:

```
(define length
  (λ (L : nlist)
    (nlist-cases L
      (λ (emptyL : emptyT) 0)
      (λ (consL : consT) (+ 1 (length (r consL)))))))
```

Splitting up datatype

Let's split up the datatype declaration of nlist into three fragments, one for each type we create:

```
(define-type nlist)

(define-type nemptyT <: nlist)

(define-type nconsT <: nlist
  [n num]
  [r nlist])
```

We could also use this form of **define-type** to describe our abstract syntax tree, which now becomes:

```
(define-type LExp)

(define-type numE <: LExp
  [n num])

(define-type addE <: LExp
  [lhs LExp]
  [rhs LExp])
```

Assume the compiler automatically creates the cases function:

$$\textit{LExp-cases} : \mathsf{LExp} \times (\mathsf{numE} \to \alpha) \times (\mathsf{addE} \to \alpha) \longrightarrow \alpha$$

So we can write our intepreter using *LExp-cases*:

```
(define eval
  (λ (e : LExp)
    (LExp-cases e
      (λ (e : numE) (n e))
      (λ (e : addE) (+ (eval (lhs e)) (eval (rhs e)))))))
```

However, nothing prevents us from adding a new subtype of LExp:

```
(define-type multE <: LExp
  [lhs LExp]
  [rhs LExp])
```

How can we write a new interpreter *eval-new* that consumes a new LExp (with the multE subtype) without rewriting the exisiting interpreter *eval*, or even looking at the source of *eval*? Try to do so as an exercise.

Where is all of this going? The **define-type** construct looks a lot like C++ or Java class declarations. In fact, C++ must use run-time tags on objects for the same reasons as we've presented here—to perform downcasting safely. In the next class, we will explore object-oriented languages further.