

First-Class Hidden Types

Joe Hallett

Sun Microsystems Laboratories and Boston University

April 17, 2006

Joint work with: Eric Allen, Victor Luchangco,
Sukyoung Ryu, Sam Tobin-Hochstadt

Subtyping

- Subtype polymorphism allows code reuse

```
class Container {  
    Object element;  
    void addElement(Object e) {this.element=e;}  
    Object getElement() {return this.element;}  
}
```

> Storing: safe **upcasts**

```
Container c = new Container();  
c.addElement(new Integer(0));
```

> Retrieving: potentially unsafe **downcasts**

```
Integer x = (Integer) c.getElement();
```

Generics

- Parametric polymorphism also allows code reuse

```
class Container<X> {  
    X element;  
    void addElement(X e) {this.element=e;}  
    X getElement() {return this.element;}  
}
```

```
Container<Integer> c =  
    new Container<Integer>();  
c.addElement(new Integer(0));  
Integer x = c.getElement();
```

Subtyping and generics

- Many interesting type relationships are possible:
 - > Extension of parametric polymorphic classes

```
class A<X> extends B<X>
```

- > Monomorphic extension of polymorphic classes

```
 $\forall X.$  class A extends B<X>
```

- > Variant subtyping

```
class A<X> extends Y> extends A<Y>
```

Subtyping and generics (cont)

- Interesting uses of types are also possible:
 - > Operations dependent on type parameters

```
class A {  
  cast<X>() : X =  
    typecase self in  
      X => self  
      else => error()  
    end  
}
```

Current restrictions on expressivity

- Restrictions on type relationships
 - > All type variables are parameters of the subtype

```
class A<X1 ... Xj> extends B<T1 ... Tk>
```

X₁ ... X_j are only type variables in T₁ ... T_k

- Restrictions on type-dependent operations
 - > Not allowed due to type erasure

Restrictive type relationships

- Cumbersome type definitions:

```
class Unit<D extends Dimension>
class Measurement<D extends Dimension,
                  U extends Unit<D>>
```

Restrictive type relationships

- Cumbersome type definitions:

```
class Unit<D extends Dimension>
class Measurement<D extends Dimension,
                  U extends Unit<D>>
```

- Inexpressible relations (Empty must be parameterized):

```
class List<X extends Object> extends Object
class Empty<X extends Object> extends List<X>
```

Restrictive type relationships

- Cumbersome type definitions:

```
class Unit<D extends Dimension>
class Measurement<D extends Dimension,
                  U extends Unit<D>>
```

- Inexpressible relations (Empty must be parameterized):

```
class List<X extends Object> extends Object
class Empty<X extends Object> extends List<X>
```

- Variant subtyping requires additional language constructs:

```
List<+Number>
List<? extends Number>
```

Idea of “where” clauses

- Type variables that are NOT parameters of the subtype being defined
 - > Elimination of extraneous type parameters
 - > Infinitely broad extensions
 - > Variant subtyping
- Allow type-dependent operations by retaining types at run time

Elimination of extraneous parameters

- Instead of:

```
trait Unit[D extends Dimension]
trait Measurement[D extends Dimension,
                 U extends Unit[D]]
```

we can write:

```
trait Unit[D extends Dimension]
trait Measurement[U extends Unit[D]]
  where {D extends Dimension}
```

Infinitely broad extensions

- We can define a single type `Empty` that is a subtype of all lists:

```
trait List[X extends Object] extends Object
```

```
object Cons[X extends Object](hd:X, tl:List[X])  
  extends List[X]
```

```
object Empty extends List[Y]  
  where {Y extends Object}
```

Variant subtyping

- We can define covariant lists without additional language constructs

```
trait List[X extends Y] extends List[Y]  
  where {Y extends Object}  
  cons(x:Y) : List[Y] = Cons[Y](x, self)
```

Variant subtyping

- We can define covariant lists without additional language constructs

```
trait List[X extends Y] extends List[Y]  
  where {Y extends Object}  
  cons(x:Y) : List[Y] = Cons[Y](x, self)
```

- The handle on `Y` allows us to write the following well-typed method call!

```
Cons[Integer](3, Empty).cons(5.7)
```

The resulting list has type: `List[Number]`

Finding witnesses

- Type inference (surrounding context)

```
trait List[X extends Y] extends List[Y]  
  where {Y extends Object}  
  cons(x:Y) : List[Y] = Cons[Y](x, self)
```

```
object Empty extends List[Z]  
  where {Z extends Object}
```

```
7 + ((Empty.cons(5)).hd)
```

Finding witnesses

- Type inference (surrounding context)

```
trait List[X extends Y] extends List[Y]  
  where {Y extends Object}  
  cons(x:Y) : List[Y] = Cons[Y](x, self)
```

```
object Empty extends List[Z]  
  where {Z extends Object}
```

```
7 + ((Empty.cons(5)).hd)
```

> A witness for Y is Integer

Can't always find witnesses

- Consider

```
trait List[X extends Y] extends List[Y]  
  where {Y extends Object}  
  canBelong(x:Object) : Boolean =  
    typecase x in  
      X => true  
      else => false  
    end  
  
object Empty extends List[Z]  
  where {Z extends Object}  
Empty.canBelong(5)
```

Can't always find witnesses

- Consider

```
trait List[X extends Y] extends List[Y]  
  where {Y extends Object}  
  canBelong(x:Object) : Boolean =  
    typecase x in  
      X => true  
      else => false  
    end  
  
object Empty extends List[Z]  
  where {Z extends Object}  
Empty.canBelong(5)
```

- In these cases we require type annotations at method invocations:

```
(Empty in List[Integer]).canBelong(5)
```

Finding witnesses at run time

- Some hidden type variables don't show up until run time

```
trait A
  f(x:Object) : Number = ...
```

```
trait B[X extends Number] extends A
  f(x:Object) : X = ...
```

```
object C extends B[Y]
  where {Y extends Number}
```

```
a : A = C
```

```
a.f(5)
```

Finding witnesses at run time (cont)

- When there's potential for ambiguity at run time we require overridings

```
trait A
  f(x:Object) : Number = ...
```

```
trait B[X extends Number] extends A
  f(x:Object) : X = ...
```

```
object C extends B[Y]
  where {Y extends Number}
  f(x:Object) : BottomType = ...
```

```
a : A = C
a.f(5)
```

Related work

- Variance annotations (Eiffel, Strongtalk, NextGen)
- Wildcards (Java 5.0)
- Scala
- Cecil

Conclusions

- Types are quantified by hidden type variables
- Allows expression of a broader range of type relationships
 - > Elimination of extraneous type parameters
 - > Infinitely broad extensions
 - > Variant type relationships
- Allows type-dependent operations
- Proved sound

joseph.hallett@sun.com

[http://research.sun.com/
projects/plrg](http://research.sun.com/projects/plrg)