Object Encodings in System F
Or: Stacking μ and $\exists$

Fabian Muehlboeck

March 21, 2012

1 Reminder: object-oriented programming

An object is the combination of some state that the object encapsulates, and some methods to access (and possibly manipulate) that state. People like to use object-oriented programming because it offers a nice way to organize a program. Several features are commonly found in most object-oriented languages, the most important of them are:

1. Encapsulation

   We want to factor our data into parts and we completely control all access to those parts, which makes it easier to guarantee certain invariants.

2. Subtyping

   We do not want additional information some objects might have to get in the way of exploiting common properties of our objects.

3. Classes and Inheritance

   We do not only want to share interfaces, but also implementations.

4. Open Recursion

   We would like to have a parameter *this* that lets us reference methods of the same class within the definition of said methods.

The following example code shows almost all the four features listed above:

```
public class Point
{
        protected int x;
        public Point(int x) { this.x = x; }
        public int getx() { return this.x; }
        public Point move(int dist)
                { return new Point(this.x + dist);}
        public Point bump() { return this.move(1); }
}

public class ScaledPoint extends Point
{
        protected int s;
        public ScaledPoint(int x, int s) { super(x); this.s = s; }
        public int gets() { return s; }
        public ScaledPoint move(int dist)
                { return new ScaledPoint(this.x + dist*s,s); }
        public ScaledPoint zoom (int s)
                { return new ScaledPoint(this.x,s); }
}
```

Subtyping is not directly shown, but any code that is written to use an instance of the Point-class should also be able to use an instance of the ScaledPoint-class.

It is easy to see what we can do with objects of those classes in Java (in fact, these classes are also Featherweight Java classes, which is something we will look at later). Now if we want fo formally reason about programs of this form, we can basically do two things: either we translate it into a calculus we already know, or we can invent a new calculus, construct all the rules we need and proof the properties we would like to have. We will use the former approach: we will encode objects and object-oriented programs in System F (plus varying extensions), which has the nice effect that most proofs are already there and we can use it to compile our program in a type-preserving way.

In the following, we will first look at some different basic object encodings for the general concept of object-oriented programming that were proposed in the 80s and 90s, until we finally look at an encoding created to support type-preserving compilation for Featherweight Java from the early 2000s.

2 Basic object encodings

All the following examples in this section use System $F^{\omega}_{<:}$ with records and pairs, some also use existential types and/or recursive types (we use letrec to construct these). The syntax and the most important rules are listed in Appendix A. For readability, we omit fold- and unfold-annotations.

2.1 Interfaces, subtyping and binary methods

We will encode our interface for a Point type as follows:

$$PT := \lambda\beta :: *.\{getx : \mathbf{Unit} \to \mathbf{Int}, move : \mathbf{Int} \to \beta, bump : \mathbf{Unit} \to \beta[, equals : \beta \to \mathbf{Bool}]\}$$

Similarly, ScaledPoint will be:

$$\begin{aligned} SPT := \lambda\beta :: *.\{ \quad & getx : \mathbf{Unit} \to \mathbf{Int}, move : \mathbf{Int} \to \beta, bump : \mathbf{Unit} \to \beta, gets : \mathbf{Unit} \to \mathbf{Int}, \\ & zoom : \mathbf{Int} \to \mathbf{Unit}[, equals : \beta \to \mathbf{Bool}]\} \end{aligned}$$

This way, if we have some τ such that $PT\tau$ is a well-formed type, any instance of that type will be a record that exposes some methods but does not expose any state (the methods however might do that). If we have such an instance p, we could for example write $p.getx$ $\mathbf{()}$ to get the x-position of the point.

The *equals*-method is put between brackets because we will always look at it separately. The reason for this is that it is a so-called binary method, i.e. a method that takes a parameter an object of the same type as the object that we are sending the message to. Having β in a contravariant position complicates subtyping a bit: an SPT (that is, a type that results from applying SPT to some argument) is a subtype of a PT if and only if SPT and PT were applied to the same argument. As we want methods of Point to return instances of Point and methods of ScaledPoint to return instances of ScaledPoint, this will usually not be the case, hence the record types that we want will not be subtypes of each other when we add a method like *equals*.

On the other hand, the type function SPT itself is a subtype of the type function PT regardless of wheter we include the *equals*-method or not, because here pointwise higher-order subtyping applies, which state that the results have to be in a subtyping relation if we apply the same argument (which is, as we saw above, true in our case). This will have some interesting consequences in the different encodings that we are going to look at.

2.2 OR - encoding objects using recursive types

When thinking about how to encode an object, one might focus on the fact that it should sometimes return an instance of its own type (as we saw in the move- and bump-methods for the Point-class). This leads to the use of recursive types, and thus our first encoding:

$$OR := \lambda\alpha :: * \rightarrow *.\mu\beta.\alpha\beta$$

OR takes a signature and produces from it a recursive type. Applied to our Point signature PT, we get

$$OR\ PT := \mu\beta.\{getx : \mathbf{Unit} \rightarrow \mathbf{Int}, move : \mathbf{Int} \rightarrow \beta, bump : \mathbf{Unit} \rightarrow \beta\}$$

Can we create an instance of that type? Yes:

$$\begin{aligned} makepoint \quad := \quad & \mathbf{letrec}\ mkobj : \mathbf{Int} \rightarrow OR\ PT = \\ & \qquad \lambda x : \mathbf{Int}. \\ & \qquad\quad \{ \quad getx = \lambda\_{:}\mathbf{Unit}.\, x, \\ & \qquad\qquad move = \lambda d{:}\mathbf{Int}.\, mkobj\ (x + d), \\ & \qquad\qquad bump = \lambda\_{:}\mathbf{Unit}.\, mkobj\ (x + 1)\} \\ & \mathbf{in}\ mkobj \end{aligned}$$

Similarly, for SPT, the type is:

$$OR\,SPT := \mu\beta.\{getx : \mathbf{Unit} \rightarrow \mathbf{Int}, move : \mathbf{Int} \rightarrow \beta, bump : \mathbf{Unit} \rightarrow \beta, gets : \mathbf{Unit} \rightarrow \mathbf{Int}, zoom : \mathbf{Int} \rightarrow \mathbf{Unit}\}$$

and an instance is created by:

$$\begin{aligned} myspoint \quad := \quad & \mathbf{letrec}\ mkobj : \{x : \mathbf{Int}, s : \mathbf{Int}\} \rightarrow OR\ SPT = \\ & \qquad \lambda p : \{x : \mathbf{Int}, s : \mathbf{Int}\}. \\ & \qquad\quad \{ \quad getx = \lambda\_{:}\mathbf{Unit}.\, p.x, \\ & \qquad\qquad move = \lambda d{:}\mathbf{Int}.\, mkobj\ \{x = p.x + d * p.s, s = p.s\}, \\ & \qquad\qquad bump = \lambda\_{:}\mathbf{Unit}.\, mkobj\ \{x = p.x + s, s = p.s\}, \\ & \qquad\qquad gets = \lambda\_{:}\mathbf{Unit}.\, p.s, \\ & \qquad\qquad zoom = \lambda d{:}\mathbf{Int}.\, mkobj\ \{x = p.x, s = s * d\}\} \\ & \mathbf{in}\ mkobj \end{aligned}$$

You might have observed a few things here:

1. Encapsulation works quite naturally. There is no way one could look at the content of x from the outside. Also we can easily call methods by just using the .-notation:

$$(((makepoint\ 5).bump\ ()).move\ 2).getx\ () = 8$$

2. Subtyping works fine (without binary methods): if we just replace *makepoint* in the above code by *makespoint*, we get the same result.

3. We did not implement the bump-method in terms of move as we did in the initial Java-code. It was not possible since there was no way to access the move-method in the definition of the bump-method.

4. While *OR SPT* is a subtype of *OR PT*, we did not use inheritance. Intuitively, this was not possible because both objects completely encapsulate their inner state.

2.2.1 Self-recursion

Let us fix the bump-method first. What we need is a self-parameter that we can use in the method definitions, such that we can write something like $bump = \lambda\_{:}\mathbf{Unit}.\ self.move\ 1$. So let's just create such a self:

$$\begin{array}{lll} makepoint & := & \mathbf{letrec}\ mkobj : \mathbf{Int} \to OR\ PT = \\ & & \quad \lambda x : \mathbf{Int}. \\ & & \quad\quad \mathbf{let}\ self\ = mkobj\ x\ \mathbf{in} \\ & & \quad\quad\quad \{\ \ getx = \lambda\_{:}\mathbf{Unit}.\ x, \\ & & \quad\quad\quad\quad move = \lambda d{:}\mathbf{Int}.\ mkobj\ (x+d), \\ & & \quad\quad\quad\quad bump = \lambda\_{:}\mathbf{Unit}.\ self.move\ 1\} \\ & & \mathbf{in}\ mkobj \end{array}$$

The same principle works for ScaledPoint.

2.2.2 Binary Methods

Now it is time to look at our *equals*-method. In Java, if we have an argument of the same class, we can access its private members, that is, we can also access the values that form its state. This is not possible in the OR encodings (in fact, it is not possible in any of the encodings presented in this part), because we do not handle the classes as types and thus do not know the internal structure of an object even if it implements some interface. Luckily, we have the *getx*-method, so we can just ask an object for that value from the outside:

$$\begin{array}{lll} makepoint & := & \mathbf{letrec}\ mkobj : \mathbf{Int} \to OR\ PT = \\ & & \quad \lambda x : \mathbf{Int}. \\ & & \quad\quad \mathbf{let}\ self\ = mkobj\ x\ \mathbf{in} \\ & & \quad\quad\quad \{\ \ getx = \lambda\_{:}\mathbf{Unit}.\ x, \\ & & \quad\quad\quad\quad move = \lambda d{:}\mathbf{Int}.\ mkobj\ (x+d), \\ & & \quad\quad\quad\quad bump = \lambda\_{:}\mathbf{Unit}.\ self.move\ 1, \\ & & \quad\quad\quad\quad equals = \lambda o{:}OR\ PT.\ x = o.getx\} \\ & & \mathbf{in}\ mkobj \end{array}$$

The problem here is that we lose subtyping. $OR\ PT$ and $OR\ SPT$ including *equals* look as follows:

$$OR\ PT := \mu\beta.\{getx : \mathbf{Unit} \to \mathbf{Int}, move : \mathbf{Int} \to \beta, bump : \mathbf{Unit} \to \beta, equals : \beta \to \mathbf{Bool}\}$$

$$\begin{array}{ll} OR\ SPT := \mu\beta.\{ & getx : \mathbf{Unit} \to \mathbf{Int}, move : \mathbf{Int} \to \beta, bump : \mathbf{Unit} \to \beta, \\ & gets : \mathbf{Unit} \to \mathbf{Int}, zoom : \mathbf{Int} \to \mathbf{Unit}, equals : \beta \to \mathbf{Bool}\} \end{array}$$

Following the Amber-rule for subtyping of recursive types, we see that $OR\ SPT$ is not a subtype of $OR\ PT$ (and of course, vice versa):

$$\frac{\alpha <: \beta \vdash F\ \alpha <: J\ \beta}{\mu\alpha.F\ \alpha <: \mu\beta.J\ \beta}\ \textsc{S-Amber}$$

2.3 OE - encoding objects using existential types

As you may have guessed, recursive types are not the only way to encode objects. Not using recursive types may make our type system strongly normalizing. Talking about interfaces and implementations, using existential types seems a natural way to encode objects, too. We present OE, a type function to create an object type from our interface:

$$OE := \lambda\alpha :: * \to *.\exists\beta.(\beta \times (\beta \to \alpha\beta))$$

Here, we need some witness type. As there is no recursion, it cannot be the object itself. Rather, we use the type of the object's state as a witness type. The type and a constructor for an object of the point class will then look as follows:

$$OE\ PT := \exists\beta.(\beta \times (\beta \to \{getx : \mathbf{Unit} \to \mathbf{Int}, move : \mathbf{Int} \to \beta, bump : \mathbf{Unit} \to \beta\}))$$

$$\begin{array}{lll} makepoint & := & \lambda n{:}\mathbf{Int}.\, \mathbf{pack}\ (\mathbf{Int}, (n, \quad \lambda x{:}\mathbf{Int}. \\ & & \qquad\qquad\qquad\qquad \{ \quad getx = \lambda\_{:}\mathbf{Unit}.\, x, \\ & & \qquad\qquad\qquad\qquad\quad\ \ move = \lambda d{:}\mathbf{Int}.\, x + d, \\ & & \qquad\qquad\qquad\qquad\quad\ \ bump = \lambda\_{:}\mathbf{Unit}.\, x + 1\})) \end{array}$$

The first thing we see is that methods do not return objects anymore, just state. This means that in this encoding, the caller has to repackage an object after having unpacked it and called whatever method he wanted. We can define an shorthand for calling methods:

$$o <= l := \mathbf{unpack}\ (\alpha, (s,\ m)) = o\ \mathbf{in}\ (m\ s).l$$

This way, given an object o of type $OE\ PT$, we can call all our three methods by writing $(o <= getx)\ ()$, $(o <= move)\ 3$ or $(o <= bump)\ ()$. However, this time all three methods return an **Int**. The caller on the other hand only knows this of the *getx*-method. He has to repackage the object after every call that returns the state of the object before he can send the next message (and he cannot do anything else with it, so we still have encapsulation here). We cannot put the repackaging code into the shorthand we defined above, because not every method returns an instance of the state type (imagine the state would rather be a record with one **Int**-field, *getx* would still return an **Int**, but *move* and *bump* would not).

We can again use shorthands, this time define them for all the methods that need repackaging, for example:

$$o << move := \lambda d{:}\mathbf{Int}.\, \mathbf{unpack}\ (\alpha, (s,\ m)) = o\ \mathbf{in}\ (\mathbf{pack}\ (\alpha, ((o <= move)\ d,\ m))\ \mathbf{as}\ OE\ PT)$$

2.3.1 Self-recursion

Self-recursion in OE works similar to OR. We have to use **letrec** to create the function that creates the method record from some state, then we can create a self instance of that record and use it in the definition of our methods. Then we package the function along with some state into the existential type - no recursion will be visible to the outside.

2.3.2 Binary Methods

There are good news and bad news related to binary methods in OE.

The good news is: subtyping works. Because of the existential outside, it is only important if the inner parts are in a subtyping relation when the witness type is the same for both types.

The bad news is: we cannot program equals (and no other binary method, for that matter). This is because the existential type parameter is not the type of the whole object, but just of the state. As we are just discussing objects that implement some interface, but have their state hidden in an existential type, we do not know what the state of any other object looks like. E.g. if we have two objects o_1, o_2 of type $OE\ PT$, we cannot write $(o_1 <= equals)\ o_2$ since the argument of equals must be of the state type of o_1. But even if we unpack o_2 to access its state, we do not know it's type. Therefore we are unable to write a well-typed binary method in our OE-setting.

2.4 ORE - encoding objects using recursive and existential types

Repackaging the object every time as the caller of a method is quite tedious. But in order to be able to return an already repackaged object, we need to know the type of the object for the return types, bringing us back to recursive types and our next encoding:

$$ORE := \lambda\alpha :: * \to *.\mu\beta.\exists\gamma.(\gamma \times (\gamma \to \alpha\beta))$$

We will skip the incremental definition an just provide types and implementations of our classes in this scheme:

$$\begin{aligned} ORE\ PT := \mu\beta.\exists\gamma.(\gamma \times (\gamma \to \{ \quad & getx : \mathbf{Unit} \to \mathbf{Int}, move : \mathbf{Int} \to \beta, bump : \mathbf{Unit} \to \beta \\ & [, equals : \beta \to \mathbf{Bool}]\})) \end{aligned}$$

$$\begin{aligned} close_{Point} := \lambda obj \quad & :\{x : \mathbf{Int}\} \times (\{x : \mathbf{Int}\} \to PT\ ORE\ PT). \\ & \mathbf{pack}\ (\{x : \mathbf{Int}\}, obj)\ \mathbf{as}\ ORE\ PT \end{aligned}$$

$$\begin{aligned} makepoint := \quad & \mathbf{letrec} \quad makemeth : \{x : \mathbf{Int}\} \to PT\ ORE\ PT = \\ & \quad \lambda p: \quad \{x : \mathbf{Int}\}. \\ & \qquad \mathbf{let} \quad self = close_{Point}\ (p.x, makemeth,)in \\ & \qquad\quad \{getx = \lambda\_{:}\mathbf{Unit}.\, p.x, \\ & \qquad\quad move = \lambda d{:}\mathbf{Int}.\, close_{Point}\ (\{x = p.x + d\},\ makemeth), \\ & \qquad\quad bump = \lambda\_{:}\mathbf{Unit}.\, self <= move\ 1\} \\ & \mathbf{in} \\ & \quad \lambda n{:}\mathbf{Int}.\, close_{Point}\ (\{x = n\},\ makemeth) \end{aligned}$$

$close_x$ is a notational shortcut for repackaging an object. In contrast to OE, binary methods can be programmed in ORE, as in OR, because the type of the argument is a whole object again, and we can express message sending uniformly as follows:

$$o <= l := \mathbf{unpack}\ (\alpha, (s,\ m)) = o\ \mathbf{in}\ (m\ s).l$$

However, as in OR, subtyping gets lost on the way.

2.5 ORBE - encoding objects using recursive and bounded existential types

$$ORBE := \lambda\alpha :: * \to *.\mu\beta.\exists\gamma <: \beta.(\gamma \times (\gamma \to \alpha\gamma))$$

The basic idea of ORBE comes from the an alternative implementation of self-recursion in OE. Instead of creating self from the state each time we create the method record from the state, we could create self from the state outside of the existential, and then just pack the whole self instead of just the state - thus the state of the object is the object itself. This could eliminate the need for re-packaging the object on the outside, but there we do not know this from the type OE constructs. ORBE can be seen as an extension of OE that makes the shape of its state at least partly public. An interesting thing is that any ORBE object could also be an ORE or OE object. Sadly, it does not combine the two in terms of binary methods in terms of being expressible and retaining subtyping, but rather it suffers from the same deficiency as OE. Subtyping with binary methods would work, but there is no way to program the function right because altough the state type is at least partially known, that still does not suffice to know that an argument exactly has the same state type as the object. A last problem of ORBE is also that in order for subtyping to work, we need to allow the bounds of the bounded existential types to vary. The rule that allows that makes the subtyping relation undecidable.

2.6 Summary

This part gave an overview of the Comparing Object Encodings paper by Bruce, Cardelli and Pierce. All the compared encodings have their strenghts and weaknesses, but it seems that the weaknesses of OR and ORE are more acceptable than those of OE and ORBE. Sadly, their comparison does not give any substantial difference between OR and ORE. The only difference they claim is that in OR, the default protection of instance variables is public since there is no existential to express that something is hidden. However, those variables are not accessible from the outside either, since they are hidden in the implementation of the methods.

3 Type-preserving compilation of Featherweight Java

3.1 Featherweight Java

Featherweight Java is a subset of real Java that provides a functional kernel of the language, designed to be easily extensible and as small as possible while still expressive. It provides classes and inheritance, but no interfaces, and a strictly functional, stateless style. Fields can be assigned only once, in the constructor, which in turn takes a value for every variable as argument. The last thing that is supported is dynamic casting. There are no ints or bools or control structures, but exactly five forms of terms: variables, field accesses, method invocations, constructor invocations and casts.

Along with FJ comes a complete yet simple type system and operational semantics with proofs of progress and preservation. We will look at a proposed encoding of FJ in System F designed for type-preserving compilation. The full details of the encoding are not within the scope of this notes - we will just look at the way objects are encoded and how subclassing and inheritance work. In general, the presented encoding is able to support type-preserving, modular compilation of full Featherweight Java.

3.2 System F extensions

The most important extension this encoding uses are row types. These are ordered records that are annotated with a set of labels that may not occur in the record, and they have an optional tail. In short, a row type looks like $\{l_1 : \tau_1, \ldots, l_n : \tau_n; \tau'\}$ and has kind R^L, where τ' is the tail of the row and has kind $R^{L'}$ s.t. $L' = L \cup \{l_1, \ldots, l_n\}$ and $\{l_1, \ldots, l_n\} \cap L = \emptyset$ (this means that L is a set of labels that may not appear in the record, therefore they also may not appear in the tail. In addition to that, the tail may not contain labels that already appear in the front of the record). The second important extension we will see are tuple kinds. They basically are records for types, written $\{l_1 :: k, ..., l_n :: k\}$. When given a type α of that kind, we can select the type labeled l_i by writing $\alpha \cdot l_i$.

3.3 Object Encoding

The object encoding that was chosen here very closely resembles the typical in-memory layout of objects: it is a row type that's first element is another row that contains the methods of the object, and the rest of the elements are the fields. Another important basic concept is that self-application is used for method calls, i.e. $o.bump$ becomes $(o.vtab.bump)o$. This means that the first parameter of every method will have the type of the object, so we already have a self/this included in our method calls. We will see that this has a rather drastic effect on subtyping.

Let us see how Point would look like in this encoding. First, we need a row type that matches the convention mentioned above: a method table at the beginning, and after that, the fields:

$$\tau_{Point_{(1)}} := \{vtab : \{getx : ?PT? \to \mathbf{Int}, move : ?PT?, \mathbf{Int} \to ?PT?, bump : ?PT? \to ?PT?\}, x : \mathbf{Int}\}$$

So we have the general structure of our type, and we know at least some input and output types. For $?PT?$ we have to insert the type of the object itself. Fortunately, we know a way how to do that:

$$\tau_{Point_{(2)}} := \mu\alpha :: *.\{vtab : \{getx : \alpha \to \mathbf{Int}, move : \alpha, \mathbf{Int} \to \alpha, bump : \alpha \to \alpha\}, x : \mathbf{Int}\}$$

That almost looks like the OR encoding that we looked at in the beginning. There is just a small problem: the α is in contravariant positions now. If we extend Point to ScaledPoint, we get:

$$\tau_{ScaledPoint_{(2)}} := \mu\alpha :: *.\{ \begin{array}{rl} vtab : \{ & getx : \alpha \to \mathbf{Int}, move : \alpha, \mathbf{Int} \to \alpha, bump : \alpha \to \alpha, \\ & gets : \alpha \to \mathbf{Int}, zoom : \alpha, \mathbf{Int} \to \alpha\}, \\ x : \mathbf{Int}, s : \mathbf{Int}\} & \end{array}$$

Here, ScaledPoint is not a subtype of Point. Luckily, we can add a thing to our encoding such that it does not need to be: the important observation is that when we check a program statically, we only care for the declared types of our variables. That is, if we supply a ScaledPoint for a variable of declared type

Point, we only care about the parts of the records that are in Point. These can be conveniently found in the front of the records, whereas the parts that ScaledPoint adds are in the tail. So all we have to do is make a ScalePoint look like a Point at certain positions in the code by hiding those tails. This can be achieved with an existential type:

$$\begin{aligned}\tau_{Point} := \exists\beta \;&:: \{f :: R^{\{vtab,x\}}, m :: * \to R^{\{getx,move,bump\}}\}.\\ &\mu\alpha :: *.\{vtab : \{getx : \alpha \to \mathbf{Int}, move : \alpha, \mathbf{Int} \to \alpha, bump : \alpha \to \alpha; \beta \cdot m\ \alpha\}, x : \mathbf{Int}; \beta \cdot f\}\end{aligned}$$

Now we just need to repackage ScaledPoint into a Point by providing the correct witness type, which in this case would be:

$$\{f = \{s : \mathbf{Int}\}, m = \lambda\alpha :: *.\{gets : \alpha \to \mathbf{Int}, zoom : \alpha, \mathbf{Int} \to \alpha\}\}$$

Note that binary methods like *equals* would still not work. The reason for this is that all the alphas are replaced with the same type on repackaging, hence the argument for the equals-method of a repackaged ScaledPoint would demand the witness type of the repackaged ScalePoint, which is not the witness type of a Point, hence we could not use a Point as an argument to the equals-method of a ScaledPoint that is repackaged as a Point, violating the substitution principle.

The last component that should be enabled by the FJ→F encoding is open recursion. We want to be able to have mutually recursive types, and possibly have types to explicitly refer to themselves. Therefore, we recursively define a tuple of types that consists of all our types:

$$\begin{aligned}\mu\gamma :: \quad &\{ \quad Point :: *, ScaledPoint :: *\}.\\ &\{ \quad Point = \exists\beta :: \{f :: ..., m :: ...\}.\mu\alpha :: *.\{vtab : ..., ...\}\\ &\quad\;\; ScaledPoint = \exists\beta :: \{f :: ..., m :: ...\}.\mu\alpha :: *.\{vtab : ..., ...\}\}\end{aligned}$$

This way, we can get a less powerful version of the *equals*-method while keeping subtyping or rather the ability to repackage ScaledPoints as Points: we can explicitly let the type of the second argument be Point (in our notation expressed as $\gamma \cdot Point$) instead of α in both classes. Then both have to accept any value packaged as Point, and different packagings have no influence on the type of the second argument.

3.4 Subclassing and Inheritance

In languages with classes, objects of a class usually share their implementation, i.e. they all have the same method table. In the presented translation, this is represented by polymorphic dictionary records. The key idea is that we accept any tails for the method and field records to be able to specify the self type. Then a subtype just has to specify the right tails to instantiate the a version of the supertype's method record that is tailored to the subtype.

We give a short example how that would look like for our running example. Here, $ktail_X$ represents the tail kind of some class X. It is a record kind of the form $\{m : * \to R^M, f : R^F\}$, where M is the set of already used method names in X and F is the set of already used field names, and $Rows_{Y,Z}$ gives us the right type of such a tail kind that specifies the methods and fields Y has that Z does not. Lastly, $\textsc{tail}[x]$ returns the tail of a given record. The last two definitions (Rows and TAIL) do not follow the paper exactly but are simplified, thus the examples below do not fully represent the true translation, but should suffice to demonstrate the underlying ideas:

$$s_{Point} := \lambda\beta :: ktail_{Point}.\mu\alpha :: *\{vtab : \{getx : \alpha \to \mathbf{Int}, move : \alpha, \mathbf{Int} \to \alpha, bump : \alpha \to \alpha; (\beta{\cdot}m)\ \alpha\}, x : \mathbf{Int}; \beta{\cdot}f\}$$

$$\begin{aligned}dictPT := \Lambda\beta :: ktail_{Point}.\{\quad &getx = \lambda this{:}s_{Point}\ \beta.\, this.x,\\ &move = \lambda(this,\ d){:}s_{Point}\ \beta \times \mathbf{Int}.\, \{vtab = this.vtab, x = this.x + d; \textsc{tail}[this]\},\\ &bump = \lambda this{:}s_{Point}\ \beta.\, this.vtab.move\ this\ 1\}\end{aligned}$$

$$s_{ScPoint} := \lambda\beta :: ktail_{ScPoint}.\mu\alpha :: *\{vtab : \{getx : \alpha \to \mathbf{Int}, ..., gets : \alpha \to \mathbf{Int}, ...; (\beta{\cdot}m)\ \alpha\}, x : \mathbf{Int}, s : \mathbf{Int}; \beta{\cdot}f\}$$

$$\begin{aligned} dictSPT := \Lambda\beta :: ktail_{ScPoint}.\{ \quad & getx = (dictPT\ Rows_{ScPoint,Point}).getx, \\ & ... \\ & gets = \lambda this{:}s_{ScPoint}\ \beta.\ this.s, \\ & ...\} \end{aligned}$$

3.5 Visibility

Currently, all fields are public - an objects type is just a record that specifies the fields, so we can access them from anywhere given an object. An existential type can help us hide private fields from everyone else. The easiest way to do this is to alter the record layout a bit, such that private fields are in an own record for each class. The record format would thus be:

$$\{vtab : ..., point : \{px : ..., py : ...\}, pz : ..., spoint : \{spx : ...\}, spy : ...\}$$

Here, a point would have private fields *px* and *py* and a public field *pz*, and a scaled point would have a private field *spx* and a public field *spy*. Now the type of the private fields would be hidden in existentials, and we would have to do some more packing and unpacking, but we would have a basic implementation of private fields. However, in Java, one may access the private fields of another object of the same class. This gets a little trickier, a lot of information about all the types in the program has to be carried around throughout the code.

Private methods on the other hand are easy - the dictionary definition just has to be extended such that it includes their definition such that the other methods can use them, but does not include them in the result.

Implementing protected and package visibilities is a lot harder, like with accessing private fields of other objects of the same class, a lot more information needs to be passed around to enable things to know the right types at the right positions.

A F-Omega - syntax, extensions and important rules

A.1 Syntax of F-Omega

$$\begin{array}{llll} \textit{Terms} & e & ::= & x \mid \lambda x{:}\tau.\, e \mid e\ e \mid \Lambda\alpha <: \tau.e \mid e[\tau] \mid (e,\, e) \mid \textbf{fst}\ e \mid \textbf{snd}\ e \\ & & & \{l = e \ldots l = e\} \mid \textbf{let}\ x = e\ \textbf{in}\ e \mid \textbf{letrec}\ x : \tau \to \tau = e\ \textbf{in}\ e \mid \\ & & & \textbf{pack}\ (\alpha, e)\ \textbf{as}\ \tau \mid \textbf{unpack}\ (\alpha, x) = e\ \textbf{in}\ e \\ \textit{Values} & v & ::= & b \mid \lambda x{:}\tau.\, e \mid \Lambda\alpha <: \tau.e \mid (v,\, v) \mid \{l = v \ldots l = v\} \mid \textbf{let}\ x = e\ \textbf{in}\ e \mid \\ & & & \textbf{letrec}\ x : \tau \to \tau = e\ \textbf{in}\ e \mid \textbf{pack}\ (\alpha, v)\ \textbf{as}\ \tau \\ \textit{Types} & \tau & ::= & B \mid \alpha \mid \tau \to \tau \mid \forall\alpha.\tau \mid \lambda\alpha :: k.\tau \mid \tau\ \tau \mid \tau \times \tau \\ & & & \exists\alpha.\tau \mid \mu\alpha.\tau \mid \{l : \tau \ldots l : \tau\} \\ \textit{Kinds} & k & ::= & * \mid k \to k \\ \textit{Var-Contexts} & \Gamma & ::= & \emptyset \mid \Gamma, x : \tau \\ \textit{Type-Contexts} & \Delta & ::= & \emptyset \mid \Delta, \alpha :: k \end{array}$$

Here, x are variables, b are base values, B are base types, and l are labels.

A.2 Important Subtyping rules

$$\frac{\alpha <: \beta \vdash F\ \alpha <: J\ \beta}{\mu\alpha.F\ \alpha <: \mu\beta.J\ \beta}\ \textsc{S-Amber}$$

The Amber rule gives us subtyping for recursive types. F and J are type functions.

$$\frac{\alpha <: \tau_1 \vdash \tau_3 <: \tau_2}{\forall \alpha <: \tau_1 . \tau_3 <: \forall \alpha <: \tau_1 . \tau_2} \textsc{S-All-Kernel-Sub}$$

This rule is called the Kernel subtyping rule for bounded polymorphic functions. It allows subtyping only for invariant bounds, and suffices for all presented encodings except for ORBE.

$$\frac{\vdash \tau_1 <: \tau_4 \qquad \alpha <: \tau_1 \vdash \tau_3 <: \tau_2}{\forall \alpha <: \tau_4 . \tau_3 <: \forall \alpha <: \tau_1 . \tau_2} \textsc{S-All-Full-Sub}$$

ORBE needs theFull subtyping rule for bounded polymorphic functions in order to be able to have variable bounds. This makes the subtyping relation undecidable.

A.3 Row types

Basic explanations of the presented extensions were given in section 3.2 . We give the typing rules for row types:

$$\frac{}{\Delta \vdash Abs^L :: R^L} \ (1)$$

Abs^L is the type of an empty row that trivially contains none of the labels in L and therefore has the corresponding row kind.

$$\frac{\Delta \vdash \tau :: * \qquad \Delta \vdash \tau' :: R^{L \cup \{l\}}}{\Delta \vdash l : \tau ; \tau' :: R^{L - \{l\}}} \ (2)$$

Next, appending a label with some type to a row type that does not already contain that label procudes a new type of a kind that does not exclude that label anymore (since it already contains it).

$$\frac{\Delta \vdash \tau :: R^{\emptyset}}{\Delta \vdash \boldsymbol{\{}\tau\boldsymbol{\}} :: *} \ (3)$$

And lastly, a row constructor (the bold braces) used with a type of a row kind that does not exclude any labels (one could also say it does not miss any labels) is viewed as complete, hence we can lift it to a full type.