Methods as pattern-matching functions

C. BARRY JAY
University of Technology, Sydney

December 16, 2003

Abstract

Methods that are specialised on sub-classes introduce a number of well-known challenges for type systems which can now be met in the *pattern calculus*. It provides a foundation for computation based on pattern-matching in which different cases may have different specialisations of a default type. Specialising by both type substitution and sub-typing supports well-typed functions whose cases correspond to the different specialisations of a method.

1 Introduction

Various calculi and type systems have been proposed as a foundation for object-oriented programming, employing a wide variety of concepts such as *bounded quantification* [CM94], the *self calculus* [AC97] and *self types* e.g. [Bru02] and [BFSvG03]. Unfortunately, none of the novel constructions introduced to handle objects have completely resolved some fundamental typing problems, as summarised in four examples by Bruce [Bru02, Chapter 3]. The heart of the matter is that a single method may be specialised in various ways that have different specialised types. The *pattern calculus* [Jay03b] (based on the constructor calculus [Jay01]) supports pattern-matching functions in which different cases may have different type specialisations of a common default. In previous work specialisation was only by type substitution. By also supporting specialisation through sub-typing, this paper shows how to resolve these fundamental typing problems. Let us consider a familiar example, of points and coloured points.

A conventional class of points can be declared by

$$\begin{array}{l}\mathsf{class\ Point} = \{\\ \quad \mathsf{position} : \mathsf{Float};\\ \quad (\mathsf{move} : \mathsf{Float} \to \mathsf{Point})\ d = \{\mathsf{Point}\ (\mathsf{this.position} + d)\}\\ \quad (\mathsf{smaller} : \mathsf{Point} \to \mathsf{Point})\ x = \{\\ \qquad \mathsf{if\ this.position} \leq x.\mathsf{position\ then\ this\ else}\ x\}\\ \}.\end{array}$$

The declaration introduces a class Point with a constructor $\mathsf{Point} : \mathsf{Float} \to \mathsf{Point}$. Objects in the class support a field position and two methods, move that moves a point and smaller that compares points. The notation used for method bodies is functional, but this is not essential. Now suppose that there is a class Colour

of colours and define a sub-class ColourPoint of Point by

```
class ColourPoint extends Point = {
  pointColour : Colour;
  (move : Float → ColourPoint) d = {
    ColourPoint (this.pointColour) (this.position + d)}
}
```

The method move has been specialised to ensure that the resulting point is coloured while smaller has been inherited.

Three typing issues arise in this example. The type of smaller on coloured points is Point → Point instead of the preferred ColourPoint → ColourPoint. That is, both the return type and the argument type have failed to adapt in line with the type of the invoking object. The third problem is how to reconcile the type of move on coloured points with that of points. A fourth problem posed in [Bru02] is how to support type parameters, e.g. to define a parametrised class of lists? Various approaches to this last problem have been proposed, e.g. [OW97, AFM97, Tho97] but have not yet been realised in practice [GJ].

Happily, type parameters play a key role in solving the three earlier problems since a type parameter can be used to represent unknown fields of possible sub-types. For example, the novel class of points is declared by

```
class Point = {
  position : Float;
  move d = {Point Null (this.position + d)}
  smaller x = {if this.position ≤ x.position then this else x}
}.
```

The syntax has only two changes, the use of Null as an extra argument to the constructor Point, and the absence of given types for the methods (which can now be inferred using a variant of Milner's algorithm $\mathcal{W}$). However, all of the typings have been implicitly parametrised. Now the type of an object in class Point has type Point T for some type T. In the simplest case T is the *top type* $\top$ whose unique value is Null. The general case is handled by using a type variable X as in Point X. Now the constructor Point has type

$$\mathsf{Point} : X \to \mathsf{Float} \to \mathsf{Point}\ X.$$

The fields and methods are interpreted as functions that act on the given object, and so have types

$$\begin{aligned}&\mathsf{position} : \mathsf{Point}\ X \to \mathsf{Float}\\ &\mathsf{move} : \mathsf{Point}\ X \to \mathsf{Float}\ \to \mathsf{Point}\ \top\\ &\mathsf{smaller} : \mathsf{Point}\ X \to \mathsf{Point}\ X \to \mathsf{Point}\ X.\end{aligned}$$

Note that the fragility of the move algorithm is exposed by its return type Point $\top$ while the robustness of smaller is revealed by its re-use of the variable X.

Now consider the sub-class of coloured points given by

```
class ColourPoint extends Point = {
  pointColour : Colour;
  move d = {Point (ColourPoint Null this.pointColour) (this.position + d)}
}
```

Note here that ColourPoint does not construct a whole point, but only the rest of the fields for a point, which must be constructed by Point.

That coloured points are points is now reflected in both the terms representing the coloured points and their types. The latter are of the form Point (ColourPoint Y) where the type variable X representing the unknown point fields has been replaced by ColourPoint Y to indicate that there is a colour field, and perhaps some others, represented by the variable Y. The constructor ColourPoint is defined to capture the new fields required by coloured points and so has type

$$\mathsf{ColourPoint} : Y \to \mathsf{Colour} \to \mathsf{ColourPoint}\ Y.$$

For example, Point (ColourPoint Null Red) 3.3 : Point (ColourPoint $\top$) is a coloured point. The approach is similar to the use of *row variables* by Remy [Gun92, Chapter 3] and by Wand [Gun92, Chapter 4] but without the overhead of introducing distinct syntactic categories for labels, records, etc.

Sub-typing can now be made *structural*. For example, every type is less than $\top$ and so Point $X <$ Point $\top$ for every type X. Hence *bounded quantifications* [CM94] such as

$$\forall(X < \mathsf{Point}).X \to \mathsf{Float}$$

can be replaced by

$$\forall X.\ \mathsf{Point}\ X \to \mathsf{Float}\ .$$

This use of type variables to capture sub-typing is reminiscent of the *phantom types* in [FP02] but without the need for bounded quantification or the need to produce abstract and concrete representations.

Now let us consider the methods. Inherited methods like smaller are *parametrically polymorphic* functions that can be applied without change to points of any kind, coloured or otherwise. This addresses two of our typing challenges. The final challenges is to type move. Its two cases can be combined into a single, pattern-matching function

```
| Point (ColourPoint x y) z →
    let this = Point (ColourPoint x y) z in
    ( | d → Point (ColourPoint Null (pointColour this) (position this + d))
| Point x y →
    let this = Point x y in
    ( | d → Point Null (position this + d))
```
(1)

in which occurrences of this have been bound to the corresponding pattern in a systematic fashion. Note that evaluation is driven by the internal structure of the object, rather than by analysing class relationships. The two cases for move above have distinct types given by Point (ColourPoint Y) $\to$ Float $\to$ Point (ColourPoint $\top$) and Point $X \to$ Float $\to$ Point $\top$.

In standard typings for pattern-matching every case must have the same type, and so the example above cannot be typed. This is because pattern-matches are typically reduced (via sums) to conditionals, whose type derivation rule is based on

$$\frac{b : \mathsf{Bool} \quad s : T \quad t : T}{\mathsf{if}\ b\ \mathsf{then}\ s\ \mathsf{else}\ t : T}\ .$$

Since the test b contains no type information both branches must take the same type. However, this process ignores the fact that the pattern p has a type P which does carry type information relevant to the specialisation.

By contrast, the pattern calculus represents pattern-matching directly using *extensions* of the form $\mathsf{at}\ p\ \mathsf{use}\ s\ \mathsf{else}\ t$ where p is the *pattern*, s is the *specialisation* and t is the *default*. Since extensions combine variable binding and branching in a single construct it provides an alternative to λ-calculus as a foundation for computation. Like the latter, its reduction rules are completely type-free, and it is able to support a variety of (related) type systems. Now let us consider how to type extensions.

In earlier typings [Jay01, Jay03b] pattern-matching was based on

$$\frac{p : P \quad s : S \quad t : T}{\mathsf{at}\ p\ \mathsf{use}\ s\ \mathsf{else}\ t : T}$$

where type specialisation for the case $\mathsf{at}\ p\ \mathsf{use}\ s\ \ldots$ was permitted but the overall type T was that of the default. It is used to define *generic functional programs* [GJ03] such as map and foldleft that can act on arbitrary data types. This rule has some structural relationship to that for extensions in EML [MBC02] but in other respects EML follows the conventional approach to classes, evaluation, sub-typing, etc. and so, despite the presence of type parameters, is unable to handle the other typing challenges above.

To handle sub-typing, the rule for extensions is here based on

$$\frac{p : P \quad s : S \quad t : T}{\mathsf{at}\ p\ \mathsf{use}\ s\ \mathsf{else}\ t : (P \to S) \wedge T}$$

where $\wedge$ is a form of type intersection indicating that the extension is both a function from P to S and of type T. Further, S is specialised by sub-typing as well as type substitution. In general, the relationship between P, S and T must be constrained to ensure type safety. Two acceptable situations are when $\sigma T = P \to S$ for some substitution σ or when T is $P \to R$ and $S < R$. In general, both type substitution and sub-typing are involved, as in the type for move given by

$$\begin{array}{c} (\mathsf{Point}\ (\mathsf{ColourPoint}\ Y) \to \mathsf{Float} \to \mathsf{Point}\ (\mathsf{ColourPoint}\ \top)) \\ \wedge \\ (\mathsf{Point}\ X \to \mathsf{Float} \to \mathsf{Point}\ \top). \end{array} \tag{2}$$

There are striking similarities between this approach and the $\lambda\&$-calculus [CGL92, Cas97] with $\mathsf{at}\ p\ \mathsf{use}\ s\ \mathsf{else}\ t$ corresponding to $\lambda p.s\ \&\ t$ and $(P \to S) \wedge T$ corresponding to $\{P \to S, T\}$. There are also fundamental differences, however. Unlike the type-free approach to pattern-matching, evaluation of terms in $\lambda\&$ is driven by explicit type information which is used to determine the best fit between the (run-time) type of the object and the possible types of the method. Also, the types of the $\lambda\&$ calculus do not include type variables, which are likely to complicate, if not invalidate, the machinery underpinning this approach. Of course, there is a large body of research on intersection types for object-oriented languages, but the relationships to this work appear to be superficial. For example, [CP96] uses intersection types (with bounded quantification) to model multiple inheritance.

The main technical contribution of this paper is an account of structural sub-typing and its use in typing extensions that solves the four key typing problems above, and provides a basis for modeling objects, classes and methods.

The remaining sections of this abstract are as follows. Section 2 introduces the untyped pattern calculus and its (Church-Rosser) reduction rules. Section 3 introduces the combinatory types and their sub-typing rules. Section 4 introduces the type derivation rules, including the separation of the patterns from the terms in general. Section 5 indicates how the fundamental concepts of object-orientation such as class, self and dynamic dispatch, can be modeled in the pattern calculus. Section 6 shows how to represent datatypes in the functional style. Section 7 draws conclusions and discusses the significant further potential of this approach. The appendix provides lemmas and the proof that reduction preserves typing.

2 The pure pattern calculus

The syntax of the *patterns* (meta-variable p) and *raw terms* (meta-variable t) of the pattern calculus is given by

$$\begin{array}{rcl} p & ::= & x \mid c \mid p\ p \\ t & ::= & x \mid c \mid t\ t \mid \mathsf{at}\ p\ \mathsf{use}\ t\ \mathsf{else}\ t \mid \mathsf{let}\ x = t\ \mathsf{in}\ t. \end{array}$$

The *variables* are represented by the meta-variable x. The *constructors* (meta-variable c) are constants of the language which do not appear at the head of any evaluation rule. Other constants may be added if desired but their evaluation rules will not be considered explicitly in the formal development. The *application* $s\ t$ applies the function s to its argument t. The novel term form is the *extension* $\mathsf{at}\ p\ \mathsf{use}\ s\ \mathsf{else}\ t$ where p is the *pattern*, s is the *specialisation* and t is the *default*. The let-term $\mathsf{let}\ x = s\ \mathsf{in}\ t$ binds x to s in t. The declaration is recursive, in that free occurrences of x in s are bound to s itself.

Extensions combine abstraction over bound variables with a branching construction. For example, the λ-abstraction $\lambda x.s$ is short-hand for the extension

$$\mathsf{at}\ x\ \mathsf{use}\ s\ \mathsf{else}\ \mathsf{err}$$

where err is some form of error term, e.g. a non-terminating expression (such as $\mathsf{let}\ x = x\ \mathsf{in}\ x$) or an exception. Also, the conditional $\mathsf{if}\ b\ \mathsf{then}\ s\ \mathsf{else}\ t$ is given by

$$(\mathsf{at}\ \mathsf{True}\ \mathsf{use}\ s\ \mathsf{else}\ \mathsf{at}\ \mathsf{False}\ \mathsf{use}\ t\ \mathsf{else}\ \mathsf{err})\ b$$

where True and False are the usual booleans. This term may also be written as

$$\begin{array}{l} \mathsf{match}\ b\ \mathsf{with} \\ \mid \mathsf{True} \rightarrow s \\ \mid \mathsf{False} \rightarrow t \end{array}$$

using the conventions that $\mathsf{match}\ b\ \mathsf{with} f$ stands for $f\ b$ and $\mid p \rightarrow s$ stands for $\mathsf{at}\ p\ \mathsf{use}\ s\ \mathsf{else}$ and that the elided final default is err.

$$
\begin{array}{rcl}
(\mathsf{at}\ x\ \mathsf{use}\ s\ \mathsf{else}\ t)\ u & > & s\{u/x\} \\
(\mathsf{at}\ c\ \mathsf{use}\ s\ \mathsf{else}\ t)\ c & > & s \\
(\mathsf{at}\ c\ \mathsf{use}\ s\ \mathsf{else}\ t)\ u & > & t\ u \quad \text{if } u \text{ cannot become } c \\
(\mathsf{at}\ p_1\ p_2\ \mathsf{use}\ s\ \mathsf{else}\ t)\ c & > & t\ c \\
(\mathsf{at}\ p_1\ p_2\ \mathsf{use}\ s\ \mathsf{else}\ t)\ (u_1\ u_2) & > & (\mathsf{at}\ p_1 \\
 & & \mathsf{use\ at}\ p_2\ \mathsf{use}\ s\ \mathsf{else\ at}\ y\ \mathsf{use}\ t\ (p_1\ y)\ \mathsf{else\ err} \\
 & & \mathsf{else\ at}\ x\ \mathsf{use\ at}\ y\ \mathsf{use}\ t\ (x\ y)\ \mathsf{else\ err\ else\ err} \\
 & &)\ u_1\ u_2 \text{ if } u_1 \text{ is constructed} \\
(\mathsf{at}\ p_1\ p_2\ \mathsf{use}\ s\ \mathsf{else}\ t)\ u & > & t\ u \text{ if } u \text{ cannot become applicative} \\
\mathsf{let}\ x = s\ \mathsf{in}\ t & > & t\{\mathsf{let}\ x = s\ \mathsf{in}\ s/x\}
\end{array}
$$

Figure 1: Reduction rules for the pattern calculus

The set of *free variables* $fv(t)$ of a raw term t are given by:

$$
\begin{array}{l}
fv(x) = \{x\} \\
fv(c) = \{\} \\
fv(s\ t) = fv(s) \cup fv(t) \\
fv(\mathsf{at}\ p\ \mathsf{use}\ s\ \mathsf{else}\ t) = fv(t) \cup (fv(s) - fv(p)) \\
fv(\mathsf{let}\ x = s\ \mathsf{in}\ t) = (fv(t) \cup fv(s)) - \{x\}.
\end{array}
$$

Note that the free variables in a pattern bind their occurrences in the specialisation. The *substitution* $s\{u/x\}$ of a term u for a variable x in term s is defined in the usual way, as are bound variables and their α-conversion. The *terms* are defined to be equivalence classes of raw terms under α-conversion.

A *constructed term* is a term whose head is a constructor, i.e. a term which is either a constructor or of the form $t_1\ t_2$ in which t_1 is constructed. Let c be a constructor. A term u *cannot become* c if it is either a constructed term other than c or an extension. A term u *cannot become applicative* if it is either a constructor or an extension.

The *basic reduction rules* of the constructor calculus are given by the relation $>$ in Figure 1. Let us consider the cases. Suppose that the pattern is a variable x. Specialisation is achieved by β-reduction, with the argument u being substituted for x in the specialisation. Suppose that the pattern is some constructor c and the argument is a constructed term u. If u is c then the specialisation is returned else the default is applied to u. Suppose that the pattern is an application $p_1\ p_2$ and the argument is a constructed term u. If u is an application $u_1\ u_2$ then specialisation tries to match p_1 with u_1 and p_2 with u_2; if either of these matches fails then evaluation reverts to applying the default to a reconstructed version of $u_1\ u_2$ (not $u_1\ u_2$ itself since this may require re-evaluation). If u is a constructor then the default is applied to it. Reduction of a let-term replaces the bound variable by its recursive definition.

A *one-step reduction* $t \to_1 t'$ is given by the application of a basic reduction to a sub-term of t. A *reduction* $t \to t'$ is given by a finite sequence of one-step reductions $t \to_1 t_1 \to \ldots \to t'$. Reduction is Church-Rosser [Jay03b].

3 The combinatory type system

A *combinatory type system* [Jay03b] has *types* (meta-variable T) given by

$$T ::= X \mid C \mid T\,T$$

consisting of *type variables* (meta-variable X), *type constants* (meta-variable C) and *type applications* ST of a type S to a type T. Constants will be introduced as required. One we will have immediate need of is the *function type constructor* $\rightarrow$ written infix as $S \rightarrow T$ (and associating to the right). The *head* of a type variable or constant is itself: the head of an application ST is the head of S.

Raw type schemes (meta-variable τ) are given by

$$\tau ::= T \mid \forall X.\tau$$

consisting of the types themselves, and the quantification of type schemes by type variables. The free and bound variables of a type or raw type scheme are defined in the usual way, as is α-conversion of bound type variables. *Type schemes* are defined to be equivalence classes of well-formed raw type schemes under α-conversion of bound variables. A type scheme is *closed* if it has no free variables. If Δ is a sequence $X_1, \ldots, X_n$ and τ is a type scheme we may write $\forall\Delta.\tau$ in place of $\forall X_1.\forall X_2 \ldots \forall X_n.\tau$. *Type substitutions*, their action and composition are defined in the usual way. The *most general unifier* of types S and T may be written $\mathcal{U}(S,T)$ while $\mathcal{U}(S,T)\uparrow$ indicates that S and T do not have any unifiers.

Now let us consider the types of methods. The use of type parameters to represent unknown fields requires a *top* type $\top$ which is a super type of every type. Similarly, to give a type to specialised methods such as move in (2) requires a form of *type intersection* represented by the constant $\wedge$ (written infix, associating to the right, and binding less tightly than $\rightarrow$). A typical specialised method m will have a type of the form

$$m : C_1(C_2(\ldots(C_nX_n)\ldots)) \rightarrow R_n \wedge \ldots \wedge C_1(C_2X_2) \rightarrow R_2 \wedge C_1X_1 \rightarrow R_1$$

where $C_n, \ldots, C_1$ represents an ascending chain in the class hierarchy. That is, in the root class represented by C_1 the type of m is $C_1X \rightarrow R_1$. Then in the sub-class represented by C_2 its type is $C_1(C_2\ X_2) \rightarrow R_2$. Again, C_3 may or may not represent a sub-class of that of C_2 but certainly represents a sub-class of the root. Abstracting from this a little, our concern is with types of the form

$$m : Q_n \rightarrow R_n \wedge \ldots \wedge Q_1 \rightarrow R_1.$$

Type safety will require that if $Q_2 = Q_1$ then R_2 is a *sub-type* of R_1 written $R_2 < R_1$. More generally, if $i > j$ and $\sigma Q_i = \sigma Q_j$ for some substitution σ then $\sigma R_i < \sigma R_j$. This is true of the type (2).

At first sight, it is natural to assume that m has type $Q_i \rightarrow R_i$ for each i but before this can happen it is necessary to "balance the accounts" vis-a-vis type variables. That is,

$$Q_n \rightarrow R_n \wedge \ldots \wedge Q_1 \rightarrow R_1 < Q_i \rightarrow R_i$$

requires that $Q_i = Q_1$. This side condition on sub-typing is captured by a relation $\lhd$ defined by mutual induction with $<$.

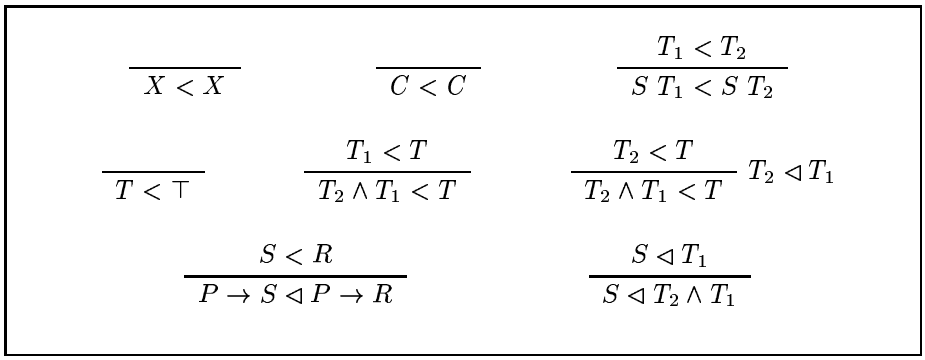

Figure 2: Sub-typing

The sub-typing rules are given in Figure 2. They are defined to ensure that sub-typing is reflexive and transitive. Sub-typing on type applications is constrained by requiring that the applied types be the same. In particular, this constraint applies to function types, i.e. $S_1 \to T_1 < S_2 \to T_2$ implies $S_1 = S_2$. That is, specialisation on argument type is limited to type variable instantiation. The rules for intersections have a side condition involving $\lhd$ as described above.

The typing of extensions will require that patterns have types that are as general as possible while being compatible with the type of the default. These various constraints are captured by the definition of the *case type* $\mathcal{V}(T, P)$ for the default type T and pattern type P given by

$$\begin{array}{l} \mathcal{V}(Q \to R \ \wedge \ T_1, P) = \\ \quad \mathsf{match}\ \mathcal{U}(P, Q)\ \mathsf{with} \\ \quad \mid \sigma \to \mathsf{if}\ \sigma S < \sigma R\ \mathsf{then}\ \mathcal{V}(T_1, P)\ \mathsf{else\ nomatch} \\ \quad \mid \uparrow \to \mathcal{V}(T_1, P) \end{array} \qquad \begin{array}{l} \mathcal{V}(Q \to R, P) = \\ \quad \mathsf{match}\ \mathcal{U}(P, Q)\ \mathsf{with} \\ \quad \mid \upsilon \to (\upsilon, \upsilon R) \\ \quad \mid \uparrow \to\ \mathsf{nomatch} \end{array}$$

with $\mathcal{V}(T, P) \uparrow$ otherwise. nomatch indicates the absence of a match, while $\uparrow$ represents failure. While case types suffice for creating the type derivation rules below, their precise nature or role is not yet clear.

4 The typed pattern calculus

The typing of the pattern calculus in Figure 3 is based on that of [Jay03b] which should be consulted for a detailed discussion. The main adjustment is to the typing of extensions which employs intersection types to capture the subtleties of sub-typing. The price for this is an additional constraint on the typing of constructors which will be addressed first.

If the constructor c has given type scheme $\forall \Delta . T_0 \to \ldots \to T_{n-1} \to T_n$ then we require that every type variable in Δ be free in T_n and that the head of T_n is a constant, the *leading type constant* of c. It follows that if X is free in T_i then it is free in T_n since the given type scheme of a constant is always closed. In order to handle sub-typing correctly, let us now impose a further constraint, that distinct constructors have distinct leading constants. When types are introduced to support classes this is a natural restriction but is inconsistent with the usual account of sum types, which must now be handled by sub-typing, as explained in Section 6.

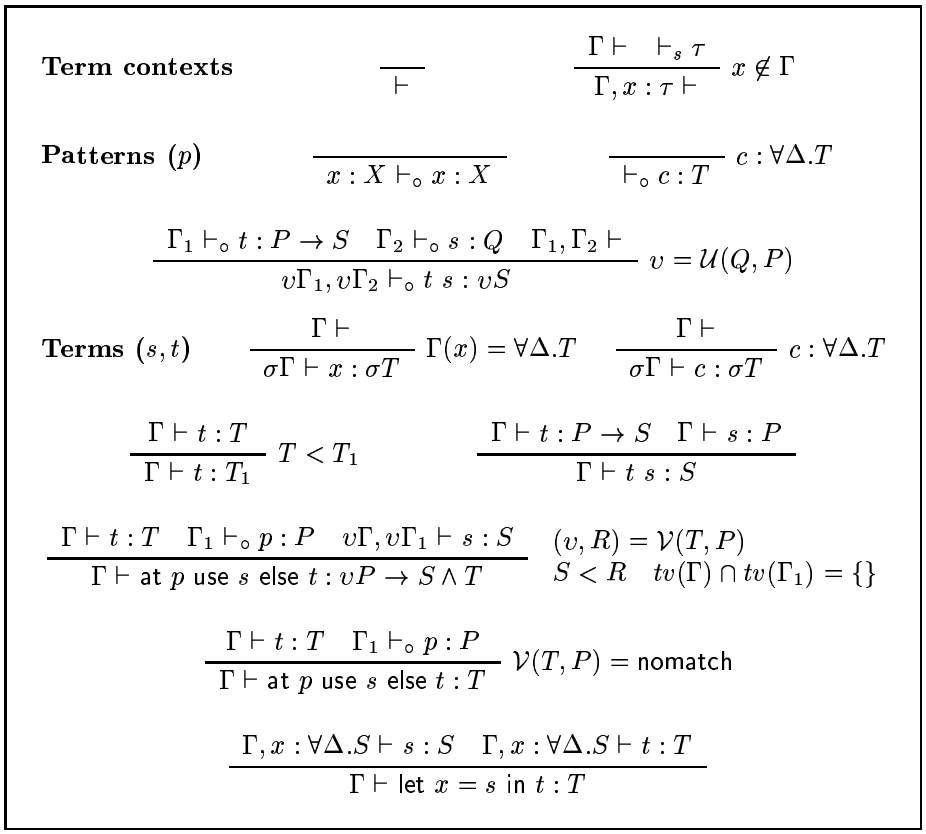

Figure 3: The typed pattern calculus

The formal type derivation rules are given in Figure 3. A *context* (meta-variable Γ) is a sequence of distinct term variables (meta-variable x) with associated type schemes. The judgement $\Gamma \vdash$ asserts that Γ is a well-formed context. The set of *free type variables* $tv(\Gamma)$ of a term context Γ contains all the free type variables of all the type schemes of all the term variables in Γ.

The *patterns* (meta-variable p) are terms built from variables, constructors and application using type derivation rules designed to ensure that patterns do not contain any repeat variables and are given their most general types. In particular, term variables are given a variable type, constructors take their most general type, and applications are typed by unifying types as necessary. The judgement $\Gamma \vdash_\circ p : P$ asserts that p is a well-formed pattern of type P in context Γ.

The judgement $\Gamma \vdash t : T$ asserts that t is a well-formed term of type T in context Γ. The derivation rules for variables and constants introduce type substitutions, but subsumption is handled by a separate rule. The rule for applications is standard. Extensions are typed by providing a type T for the default in context Γ and a type P for the pattern in completely separate context Γ_1. If the case type $\mathcal{V}(T, P)$ exists and is (υ, R) then the specialisation s must have type $S < R$ in a context specialised by υ. In providing support for type inference this is the only case of interest, but unfortunately, substitution may

cause unification to fail, so that $\mathcal{V}$ becomes nomatch. In this case, the specialisation can never be accessed and so its well-formedness becomes irrelevant. This property is captured by the second rule for typing extensions.

The let-declarations are a little unusual in that they support implicit, polymorphic recursion. That recursion is implicit is important for supporting dynamic dispatch by adding cases to existing functions. Polymorphic recursion is useful when defining functions that are generic over many data types.

Theorem 4.1 *Reduction preserves typing.*

The proof is in the appendix.

5 Interpreting classes

There appears to be a large gap between the pattern calculus, with its emphasis on (pattern-matching) functions and its Church-Rosser reduction rules, and the notion of an object responsible for its own state and behaviour. This section will show how the behaviour of objects can be captured, including the notion of "self" and dynamic dispatch; the treatment of state is pending.

A distinguishing property of object-oriented languages is the use of *dynamic dispatch*, that in the invocation $o.m$ of a method m by an object o it is o that determines the meaning of m rather than the program environment in which the invocation was made. A natural means of implementing this is to store the methods with the objects or, in a class-based language, to store the method values in a run-time representation of the class. With the introduction of subclasses, the same method may have many different algorithms, indexed first by class and then by method name. An equivalent approach is to index by method name and then by class. The approach adopted here is to index by method name, but then to return a pattern-matching algorithm whose cases correspond to the relevant classes. Dynamic dispatch is respected since pattern-matching against the invoking object determines which case is used. Note that classes do not appear as part of the run-time system. Of course, one is free to implement using classes if desired, e.g. to maintain separate compilation of classes, but the standard for typing and execution is not class-based.

The pattern-matching program for a method is generated incrementally as various (sub-)classes are declared. Let us examine this in a little more detail. Each declaration of a root class C introduces C as a type with a single constructor C of the same name. Fields and methods are given by pattern-matching against a pattern p obtained by fully applying C to some distinct variables, one for each field and one for the unknown sub-fields, i.e. "the rest". Self-reference is given by the variable this which is implicitly bound to p. Method invocation $o.m$ calling the method m of the object o is given by function application $m\ o$.

When a sub-class S of C is declared then S becomes a type and constructor just as before, the arguments being given by the new fields in S and its "rest". When a method m is specialised in S then the current global value t of m is replaced by an extension at q use s else t where s is the new method body and q is a pattern for objects in class S built using both of the constructors C and S. Since the constructor S is new-minted, the meaning of m for existing objects is unchanged, the new case only affecting objects in class S.

A final example of a class declaration is for a parametric class of list nodes given by

$$\begin{array}{l}\mathsf{class\ Node}\ X = \{\\ \quad \mathsf{val} : X;\\ \quad \mathsf{next} : \mathsf{Node\ Rest}\ X;\\ \}\end{array}$$

It introduces a type, its constructor and some functions as before, but now, for the first time, the type parameter Rest which is implicitly bound to the type of the additional fields appears in the type of a field (just as this is implicitly bound to the object). For example, next has type

$$\mathsf{next} : \mathsf{Node\ Rest}\ X \to \mathsf{Node\ Rest}\ X.$$

This account of Node supports the definition of parametrically polymorphic functions such as mapping on lists; fully generic mapping over arbitrary classes as in [Jay03b] can be anticipated.

The interaction between type parameters and sub-typing can be seen by considering a sub-class of doubly-linked lists are given by a sub-class

$$\begin{array}{l}\mathsf{class\ doubleNode\ extends\ Node}\ X = \{\\ \quad \mathsf{previous} : \mathsf{Node\ (doubleNode\ Rest)}\ X;\\ \}\end{array}$$

with constructor

$$\mathsf{doubleNode} : \mathsf{Rest}\ X \to \mathsf{Node\ (doubleNode\ Rest)}\ X \to \mathsf{doubleNode\ Rest}\ X.$$

6 Interpreting data types

While the types associated to classes need only have one constructor, reflecting the fixed choices of fields, typical datatype definitions allow for alternatives, whose semantics are given by coproduct types. For example, the data type declaration

$$\mathtt{datatype}\ \mathsf{List}\ X = \mathsf{Nil} \mid \mathsf{Cons\ of}\ X\ \mathsf{and\ List}\ X$$

typically introduces the type constant List and constructors

$$\begin{array}{rcl}\mathsf{Nil} & : & \mathsf{List}\ X\\ \mathsf{Cons} & : & X \to \mathsf{List}\ X\ \to \mathsf{List}\ X\end{array}$$

However, in the typing of the pattern calculus considered here, Nil and Cons cannot both be constructors since their result types have the same head List. The issues are more easily explored in the fundamental example of booleans, whose two values True and False are typically considered to be constructors.

Without this restriction on the types of constructors we could infer the type

$$f = \ \mid \mathsf{True} \to s \mid \mathsf{False} \to r : \mathsf{Bool} \to S \wedge \mathsf{Bool} \to R$$

given that $s : S$ and $r : R$ and $S < R$. However, $\mathsf{Bool} \to S \wedge \mathsf{Bool} \to R$ is a sub-type of $\mathsf{Bool} \to S$ and so it would follow that f False has type S even though it reduces to $r : R$.

The solution is to exploit sub-typing to separate the alternatives. For example, introduce constants

$$\begin{array}{rcl} \mathsf{HostBool} & : & X \to \mathsf{HostBool}\ X \\ \mathsf{TrueBool} & : & \mathsf{TrueBool} \\ \mathsf{FalseBool} & : & \mathsf{FalseBool} \end{array}$$

and define the type Bool of booleans to be HostBool $\top$ with

$$\begin{array}{rcl} \mathsf{True} & = & \mathsf{HostBool\ TrueBool} : \mathsf{HostBool\ TrueBool} < \mathsf{Bool} \\ \mathsf{False} & = & \mathsf{HostBool\ FalseBool} : \mathsf{HostBool\ FalseBool} < \mathsf{Bool}. \end{array}$$

Now at False use r else err : HostBool FalseBool $\to R \wedge$ Bool $X \to R$ and so

$$f : \mathsf{HostBool\ TrueBool} \to S \wedge \mathsf{HostBool\ FalseBool} \to R \wedge \mathsf{Bool}\ X \to R$$

or, by subsumption (and substitution) $f : \mathsf{Bool} \to R$.

7 Conclusions

The combination of type variable quantification and sub-typing is powerful enough to type both inheritance by parametric polymorphism and method specialisation by type specialisation, provided that the type derivation rules for pattern-matching are sufficiently liberal. In particular, the four typing challenges summarised in [Bru02] (parametricity in argument and result types, sub-typing on return types, and parametrised types for given fields) can all be met. Dynamic dispatch is handled by global update of existing function with new cases that involve new constructors. The self object becomes an implicit parameter in function definitions. The type of self is handled by a type parameter representing any unknown fields.

Now that the fundamental typing obstacles are removed, the next step is to define a complete object-oriented language in this way, preferably including the representation of state. Here are some comments on its anticipated properties.

- Access to methods of the super class can be achieved by ignoring the first match in a pattern-match.
- Multiple inheritance of classes cannot be supported in this system since the order of constructors in an object is fixed.
- A naive implementation of the pattern calculus would store all cases for a method together, instead of storing all methods for a class. However, there is nothing to stop classes being compiled separately, providing the naive semantics is respected.
- The evaluation rules pick out the first pattern that fits the argument, but when the pattern-match represents a method then the first fit is always the best fit since sub-classes must be declared after super-classes.
- Multi-methods can be easily supported.

- Algorithms to determine whether a pattern-matching covers all possible patterns for a type can be anticipated, but none have yet been developed for this calculus.

The consequences of having such a language may prove to be dramatic. First, the typed pattern calculus is significantly simpler than existing practice (let alone the various theoretical alternatives) since sub-typing is structural, classes are banished from the run-time system and evaluation is type-free. Second, wholly novel forms of expressive power are likely when this work is combined with generic functions for operations such as mapping and traversing to provide, say, a generic method for deep cloning, or *visitors* [PJ98]. Our prototype implementation already supports both suites of features in isolation. Third, the typed pattern calculus may prove to be a natural foundation for typed programming in general including such things as a generic function for in-place update [JLN02]. These ideas may lead to a calculus able to support all of the key concepts from imperative, functional and object-oriented languages in a simple unified foundation.

References

[AC97] M. Abadi and L Cardelli. *A Theory of Objects.* Monographs in Computer Science. Springer, 1997.

[AFM97] Ole Agesen, Stephen N. Freund, and John C. Mitchell. Adding type parameterization to the java language. In *Proceedings of the 1997 ACM SIGPLAN conference on Object-oriented programming systems, languages and applications*, pages 49–65, 1997.

[BFSvG03] Kim B. Bruce, Adrian Fiech, Angela Schuett, and Robert van Gent. Polytoil: A type-safe polymorphic object-oriented language. *ACM Trans. on Progr. Lang. and Sys.*, 25(2):225–290, 2003.

[Bru02] Kim Bruce. *Foundations of Object-Oriented Languages: Types and Semantics.* The MIT Press, 2002.

[Cas97] Giuseppe Castagna. *Object-Oriented Programming: A Unified Foundation.* Progress in Theoretical Computer Science Series. Birkäuser, Boston, 1997.

[CGL92] G. Castagna, G. Ghelli, and G. Longo. A calculus for overloaded functions with subtyping. In *ACM Conference on LISP and Functional Programming*, pages 182–192, 1992. Extended and revised version in *Information and Computation* 117(1):115-135, 1995.

[CM94] Luca Cardelli and John C. Mitchell. Operations on records. In C. A. Gunter and J. C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*, pages 295–350. The MIT Press, Cambridge, MA, 1994.

[CP96] Adriana B. Compagnoni and Benjamin C. Pierce. Higher-order intersection types and multiple inheritance. *Mathematical Structures in Computer Science*, 6(5):469–501, 1996.

[FP02] M. Fluet and R. Pucella. Phantom types and subtyping. In *Proceedings of the 2nd IFIP International Conference on Theoretical Computer Science (TCS 2002)*, pages 448–460, 2002.

[GJ] Generic java. `http://www.cis.unisa.edu.au/~pizza/gj/`.

[GJ03] Jeremy Gibbons and Johan Jeuring, editors. *Generic Programming: IFIP TC2/WG2.1 Working Conference on Generic Programming July 11-12,2002, Dagstuhl, Germany.* Kluwer Academic Publishers, 2003.

[Gun92] C. Gunter. *Semantics of Programming Languages.* Foundations of Computing. MIT, 1992.

[Jay01] C.B. Jay. Distinguishing data structures and functions: the constructor calculus and functorial types. In S. Abramsky, editor, *Typed Lambda Calculi and Applications: 5th International Conference TLCA 2001, Kraków, Poland, May 2001 Proceedings*, volume 2044 of *Lecture Notes in Computer Science*, pages 217–239. Springer, 2001.

[Jay03a] C.B. Jay. The constructor calculus (revised). `http://www-staff.it.uts.edu.au/~cbj/Publications/constructors3.ps`, 2003. (renamed as "The pattern calculus").

[Jay03b] C.B. Jay. The pattern calculus. `http://www-staff.it.uts.edu.au/~cbj/Publications/pattern_calculus.ps`, 2003. (accepted for publication by ACM Trans. on Progr. Lang. and Sys.).

[JLN02] C.B. Jay, H.Y. Lu, and Q.T. Nguyen. The polymorphic imperative. `http://www-staff.socs.uts.edu.au/~cbj/Publications/imperatives.ps`, 2002.

[MBC02] Todd Millstein, Colin Bleckner, and Craig Chambers. Modular typechecking for hierarchically extensible datatypes and functions. In *The 2002 International Conference on Functional Programming (ICFP 2002), Pittsburgh, PA, October 4-6, 2002*, 2002. An earlier version of this paper appeared in the Ninth International Workshop on Foundations of Object-Oriented Languages (FOOL 9), Portland, Oregon, January 19, 2002.

[OW97] Martin Odersky and Philip Wadler. Pizza into java: Translating theory into practice. In N. Jones, editor, *Proceedings 24th ACM Symposium on Principles of Programming Languages, Paris, France, January 1997*, pages 146–159. ACM, 1997.

[PJ98] Jens Palsberg and C. Barry Jay. The essence of the visitor pattern. In *Proceedings of COMPSAC'98, 22nd Annual International Computer Software and Applications Conference*, pages 9–15, Vienna, Austria, August 1998.

[Tho97] Kresten Krab Thorup. Genericity in Java with virtual types. *Lecture Notes in Computer Science*, 1241:444–471, 1997.

8 Appendix: the proof of type preservation

Lemma 8.1 *Let σ be a type substitution and let S and T be types. If $S < T$ then $\sigma S < \sigma T$. If $S \lhd T$ then $\sigma S \lhd \sigma T$.*

Proof Both statements are proved by a single induction over the length of the derivation of the premise. □

Lemma 8.2 *Let σ be a type substitution and T and P be types. If $\mathcal{V}(\sigma T, P) = (\upsilon_1, S_1)$ then $\mathcal{V}(T, P)$ is of the form (υ, S) where $\upsilon_1 \sigma = \rho\upsilon$ for some substitution ρ such that $\rho S = S_1$.*

Proof The proof is by induction on the structure of T. If T is a function type $Q \to R$ then $\sigma_1 = \mathcal{U}(\sigma Q, P)$ and so $\upsilon = \mathcal{U}(Q, P)$ exists and $\sigma_1\sigma = \rho\upsilon$ as required. If T is an intersection $Q \to R \wedge T_2$ then $\mathcal{V}(\sigma T_2, P) = (\sigma_1, S_1)$ and so $\mathcal{V}(T_2, P) = (\upsilon, S)$ exists and is $\mathcal{V}(T, P)$ where $\sigma_1\sigma = \rho\upsilon$ for some substitution ρ as required. □

Lemma 8.3 *Type derivations are stable under type substitution.*

Proof The proof is by induction on the length of the derivation. For subsumption, use Lemma 8.1. For extensions use Lemma 8.2. □

Lemma 8.4 *From derivations of $\Gamma_1 \vdash_\circ p : P$ and $\Gamma, \sigma\Gamma_1 \vdash s : S$ can be constructed a derivation of $\Gamma \vdash$ at p use s else err $: \sigma P \to S$. In particular, from a derivation of $\Gamma, x : P \vdash s : S$ can be constructed a derivation of $\Gamma \vdash \lambda x.s : P \to S$.*

Proof We can infer err $: \sigma P \to S$ and then $\mathcal{V}(\sigma P \to S, P) = (\sigma, S)$ which yields the result. □

Lemma 8.5 *Typings of terms are stable under term substitution. That is, if there are derivations $\Gamma, x : U \vdash s : S$ and $\Gamma \vdash u : U$ then there is a derivation of $\Gamma \vdash s\{u/x\} : S$.*

Proof The proof is by induction on the structure of s. □

Lemma 8.6 *If $\mathcal{V}(T, P) = (\upsilon, S)$ and $\upsilon P \to S \wedge T < Q \to R$ then there is a type $Q_0 \to R_0$ such that $T < Q_0 \to R_0$ and $\mathcal{V}(Q_0 \to R_0, P) = (\upsilon, S)$ and $\upsilon P \to S \wedge Q_0 \to R_0 < Q \to R$.*

Proof The proof is by induction on the structure of T. If T is a function type then we are done. Without loss of generality it is an intersection type $T_1 \wedge T_2$. Since $\mathcal{V}(P, T)$ is defined, it follows that T_1 is a function type, say, $Q_1 \to R_1$ and $\mathcal{V}(T_2, P) = \mathcal{V}(T, P)$. Suppose that $T < Q \to R$. If $T_2 < Q \to R$ then $\upsilon P \to S \wedge T_2 < Q \to R$ and now induction yields the result. Otherwise $Q_1 \to R_1 \lhd T_2$ and $Q_1 \to R_1 < Q \to R$. By the definitions of $\lhd$ and $\mathcal{V}$ we may assume that $T_2 = Q_2 \to R_2$. Hence $Q_1 = Q_2$ (since $Q_1 \to R_1 \lhd Q_2 \to R_2$) and so $\mathcal{V}(Q_1 \to R_1, P) = (\upsilon, S)$ (since $(\upsilon, S) = \mathcal{V}(T, P)$).

If it is not the case that $T < Q \to R$ then $\upsilon P \to S \lhd T$ and $\upsilon P \to S < Q \to R$. Then T may be replaced by T_2 and induction applied. □

Theorem 8.7 *Reduction preserves typing.*

Proof The proof is by case analysis on the basic reductions in Figure 1 since the result extends to arbitrary reductions by Lemma 8.5. Similarly, reduction of let-terms preserves typing since typing is stable under type substitution (Lemma 8.3).

Consider the typing of an application (at p use s else t) u. If the case type exists then it takes the form

$$\frac{\dfrac{\dfrac{\Gamma \vdash t : T \quad \Gamma_1 \vdash_\circ p : P \quad \upsilon\Gamma, \upsilon\Gamma_1 \vdash s : S}{\Gamma \vdash \mathsf{at}\ p\ \mathsf{use}\ s\ \mathsf{else}\ t : \upsilon P \to S \wedge T}}{\Gamma \vdash \mathsf{at}\ p\ \mathsf{use}\ s\ \mathsf{else}\ t : Q \to R} \quad \Gamma \vdash u : Q}{\Gamma \vdash (\mathsf{at}\ p\ \mathsf{use}\ s\ \mathsf{else}\ t)\ u : R}$$

where $(\upsilon, S_0) = \mathcal{V}(T, P)$ and $S < S_0$ and $\upsilon P \to S \wedge T < Q \to R$. By Lemma 8.6, T may be asumed to be a function type $Q_1 \to R_1$. Now let us consider the possible patterns and associated reductions.

Let p be a variable x. Then Γ_1 is some $x : X$. Also β-reduction produces $s\{u/x\}$ so it suffices, by Lemma 8.5, to find a derivation of $\Gamma, x : Q \vdash s : R$. Now there is a derivation of $\Gamma, x : \upsilon X \vdash s : S$ since the domain of υ is just X and now $\upsilon X = Q_1 = Q$.

Let p and u both be some constructor c. Then reduction produces s and so it suffices to establish a derivation of $\Gamma \vdash s : R$. Observe that we have $\upsilon\Gamma \vdash s : S$ since Γ_1 is empty. Observe that $c : Q$ implies that Q is ρP for some substitution ρ. If $Q_1 \to R_1 < Q \to R$ then $Q_1 = Q = \rho P$ and so $\upsilon\Gamma = \Gamma$ and $S = R_1 < R$. Alternatively, if $\upsilon P \to S \lhd Q_1 \to R_1$ and $\upsilon P \to S < Q \to R$ then $\upsilon P = Q_1$ and so $\upsilon\Gamma = \Gamma$. Also, $S < R$. Either way, the desired derivation is obtained by subsumption.

Now let p be some constructor c and u be some term that cannot become c. If u is a constructed term then P and Q have no unifiers and so $Q_1 \to R_1 < Q \to R$. Hence $\Gamma \vdash t\ u : R$ as required. A similar argument applies when p is an application $p_1\ p_2$ and u cannot become applicative.

Now let p be an application $p_1\ p_2$ and u be an application $u_1\ u_2$ where u_1 is constructed. The type derivation for the pattern is of the form

$$\frac{\Gamma_0 \vdash_\circ p_1 : P_3 \to P_0 \quad \Gamma_2 \vdash_\circ p_2 : P_2}{\upsilon_0\Gamma_0, \upsilon_0\Gamma_2 \vdash_\circ p_1\ p_2 : \upsilon_0 P_0} \quad \upsilon_0 = \mathcal{U}(P_2, P_3).$$

If $Q_1 \to R_1 < Q \to R$ then $Q_1 = Q$ and $R_1 < R$.

We can derive types for the sub-terms appearing in the result of the specialisation as follows (using Lemma 8.4) where appropriate):

$$\begin{array}{l}
\upsilon_1\Gamma, \upsilon_1\Gamma_0 \vdash f = \lambda y.\ t\ (p_1\ y) : \upsilon_1 P_3 \to \upsilon_1 R_1 \\
\upsilon_1\Gamma, \upsilon_1\Gamma_0 \vdash g = \mathsf{at}\ p_2\ \mathsf{use}\ s\ \mathsf{else}\ f : \upsilon\upsilon_0 P_3 \to S\ \wedge\ \upsilon_1 P_3 \to \upsilon_1 R_1 \\
\Gamma \vdash h = \lambda x.\lambda y.t\ (x\ y) : T' \\
\Gamma \vdash k = \mathsf{at}\ p_1\ \mathsf{use}\ g\ \mathsf{else}\ h : (\upsilon_1 P_3 \to \upsilon_1 P_0) \to (\upsilon\upsilon_0 P_3 \to S \wedge \upsilon_1 P_3 \to \upsilon_1 R_1) \wedge T'
\end{array}$$

where $\upsilon_1 = \mathcal{U}(P_0, Q_1)$ and T' is $(Y \to Q_1) \to Y \to R_1$.

Now consider the type of the original extension applied to a constructed term $u_1\ u_2$. If $Q_1 \to R_1 < Q \to R$ then $\Gamma \vdash k\ u_1\ u_2 : R_1$ and we are done. If $\upsilon P \to S < Q_1 \to R_1$ and $\upsilon P \to S < Q \to R$ then υ unifies everything desired and so $\Gamma \vdash k\ u_1\ u_2 : S < R$.

□