

The Pattern Calculus

C. BARRY JAY
University of Technology, Sydney

November 19, 2003

Abstract

There is a significant class of operations such as mapping that are common to all data structures. The goal of generic programming is to support these operations on arbitrary data types without having to re-code for each new type. The constructor calculus and combinatory type system reach this goal by representing arbitrary data structures by names and a finite set of constructors. These can be used to define generic functions by pattern-matching programs in which each pattern has a different type. Evaluation is type-free. Polymorphism is captured by quantifying over type variables that represent unknown structures. A practical type inference algorithm is provided.

1 Introduction

All data structures share a significant class of operations, such as equality, addition or traversing a structure for its data. The goal of *generic programming* [BS98, Jeu00, GJ03] is to support these operations on arbitrary data types without having to re-code for each new type. This paper reaches the goal by creating a system that:

- supports a more flexible approach to pattern-matching, grounded in the *pattern calculus*;
- expresses polymorphism in the choice of structure by quantifying over variables for higher types in the *combinatory type system*;
- expresses polymorphism in the number of sorts of data in a structure by quantifying over variables representing tuples of types;
- represents arbitrary data structures as combinations of names and a small set of constructors; and
- defines generic functions by pattern-matching over these constructors.

Also, it:

- evaluates terms without manipulating types;
- supports a practical type inference algorithm; and
- has been realised in the programming language FISH2.

1.1 Pattern-matching

Pattern-matching is a means of defining functions that act on data types. For example, given list types defined by

$$\text{type List } X = \text{Nil} \mid \text{Cons of } X \text{ and List } X$$

we can define the length of a list by

$$\begin{aligned} \text{length} : \text{List } X \rightarrow \text{Int} = \\ & \mid \text{Nil} \rightarrow 0 \\ & \mid \text{Cons } h \ t \rightarrow \text{length } t + 1 \end{aligned}$$

Similarly, given binary tree types defined by

$$\text{type Btree}_0 X = \text{Leaf}_0 \mid \text{Node}_0 \text{ of } X \text{ and Btree}_0 X \text{ and Btree}_0 X$$

we can define their sizes by

$$\begin{aligned} \text{sizetree} : \text{Btree}_0 X \rightarrow \text{Int} = \\ & \mid \text{Leaf}_0 \rightarrow 1 \\ & \mid \text{Node}_0 \ x \ y \ z \rightarrow 1 + (\text{sizetree } y) + (\text{sizetree } z) \end{aligned}$$

Generic programming aims to subsume these into a single function `size` that will compute the size of an arbitrary data structure. One way to begin would be to simply collect all of the patterns given so far into a single pattern-match, as in

$$\begin{aligned} \text{lengthsize} : X \rightarrow \text{Int} = \\ & \mid \text{Nil} \rightarrow 0 \\ & \mid \text{Cons } h \ t \rightarrow \text{lengthsize } t + 1 \\ & \mid \text{Leaf}_0 \rightarrow 1 \\ & \mid \text{Node}_0 \ x \ y \ z \rightarrow 1 + (\text{lengthsize } y) + (\text{lengthsize } z) \end{aligned}$$

Although the evaluation of this program is perfectly straightforward, it will not type-check in standard functional languages such as ML [MT91, Cam97] and Haskell [Has02] or in the *typed pattern calculus* of [?] since they require that every case in a pattern-match has the same type. This constraint is unnecessarily tight, however, as it suffices that each case specialise a default type to one appropriate to its pattern. For example, in `lengthsize` the default type $X \rightarrow \text{Int}$ instantiates X to be either `List Y` or `Btree0 Y`.

This typing is achieved by a new term form, the *extension*

$$\text{at } p \text{ use } s \text{ else } t$$

which extends the *default function* t at the *pattern* p by the *specialisation* s . The main type derivation rule for extensions, when stripped of its contexts, is

$$\frac{t : T \rightarrow T_2 \quad p : T_1 \quad s : vT_2}{\text{at } p \text{ use } s \text{ else } t : T \rightarrow T_2} v = \mathcal{U}(T_1, T)$$

where v is the most general unifier of the type T_1 of the pattern and the argument type T of the default function. The specialisation need only have type vT_2 since it will only be evaluated when T_1 and T have been unified.

The expressive power of the resulting *pattern calculus* is determined by that of the patterns. When the pattern is a variable x then the extension behaves like a λ -abstraction and specialisation is a form of β -reduction given by the reduction rule

$$(\text{at } x \text{ use } s \text{ else } t) t_1 > s\{t_1/x\}$$

in which t_1 is substituted for x in s . When the pattern is a *constructor* c then the specialisation rule is

$$(\text{at } c \text{ use } s \text{ else } t) c > s.$$

Taking c to be `True` yields a form of conditional. The most interesting cases are when patterns are given by applications $p p_1$. When such an extension is applied to an applicative term $t_1 t_2$ then specialisation attempts to match p_1 with t_1 and p_2 with t_2 (as described in Figure 4).

Patterns of the form $c p_1 \dots p_n$ for some constructor c suffice for defining functions on particular types, such as `length`, but generic functions acting on arbitrary data types require non-standard patterns of the form $x y$ where x and y are both variables. For example, the generic function

$$\begin{aligned} \text{bulk} : X \rightarrow \text{Int} = \\ | x y \rightarrow (\text{bulk } x) + (\text{bulk } y) \\ | x \rightarrow 1 \end{aligned}$$

measures the total bulk of its argument, i.e. the total number of constructors appearing in it. Note that each recursive call to `bulk` takes a different type from the others. This *polymorphic recursion* is a common feature of generic programs.

Unlike `bulk`, most generic functions require detailed information about the internal structure of their arguments to infer, for example, that to map a function f over a list `Cons h t` is to *apply* f to h and to *map* f over t .

1.2 Typing generic functions

Before considering the algorithm for mapping let us consider its type. A first attempt is

$$\text{map}^1 : (X \rightarrow Y) \rightarrow FX \rightarrow FY.$$

Here F is a variable representing a higher type, such as `List`, which determines a collection of possible structures or *shapes* [Jay95, Jay96] just as X and Y are variables able to represent any sort of data. Unfortunately, this typing is not generous enough to handle mapping over types containing more than one sort of data. For example, consider a type of binary trees defined by

$$\begin{aligned} \text{type Btree}_2 X_1 X_2 = \\ \text{Leaf}_2 \text{ of } X_1 \\ | \text{Node}_2 \text{ of } X_2 \text{ and Btree}_2 X_1 X_2 \text{ and Btree}_2 X_1 X_2. \end{aligned}$$

To map a pair of functions over such a binary tree would require some function

$$\text{map}^2 : (X_1 \rightarrow Y_1) \rightarrow (X_2 \rightarrow Y_2) \rightarrow FX_1 X_2 \rightarrow FY_1 Y_2.$$

Types with additional data parameters would require `map`³ etc. Further, all these different mapping functions are interdependent, as can be seen by considering the *uniform trees* declared by

$$\text{type Btree}_1 X = \text{Uniform of Btree}_2 X X$$

since mapping a single function across $\text{Btree}_1 X$ reduces to mapping a pair of functions across $\text{Btree}_2 X X$.

The solution adopted here is to collect all of the *data parameters* of a type declaration into a *tuple* of types such as the pair (X_1, X_2) in the declaration

```
type Btree( $X_1, X_2$ ) =
  Leaf of  $X_1$ 
  | Node of  $X_2$  and Btree( $X_1, X_2$ ) and Btree( $X_1, X_2$ ).
```

Further, tuples of functions can be represented by a single term using the *arrow* type A declared by

```
type A has
   $AXY = \text{Onefun } X \rightarrow Y$ 
  or  $A(X_1, X_2)(Y_1, Y_2) = \text{Bthfun } A(X_1, Y_1) \text{ and } A(X_2, Y_2)$ .
```

The syntax here introduces a new type A with constructors $\text{Onefun} : (X \rightarrow Y) \rightarrow AXY$ and $\text{Bthfun} : A(X_1, Y_1) \rightarrow A(X_2, Y_2) \rightarrow A(X_1, X_2)(Y_1, Y_2)$. Note that, unlike more familiar type declarations, the arguments of A differ for each constructor, even as to the number of types appearing. That is, A can be viewed as a type which has a *polymorphic kind* [Jay01] such as $\forall n. n \rightarrow n \rightarrow *$.

Then mapping on lists has type $AXY \rightarrow \text{List}X \rightarrow \text{List}Y$ and mapping on binary trees has type $A(X_1, X_2)(Y_1, Y_2) \rightarrow \text{Btree}(X_1, X_2) \rightarrow \text{Btree}(Y_1, Y_2)$. Generalising these types and quantifying over the type variables yields the type scheme for generic mapping,

$$\text{map} : \forall F, X, Y. AXY \rightarrow FX \rightarrow FY.$$

Now let us consider the type system which supports such types and schemes.

1.3 Combinatory types

The combinatory types

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

consist of type variables, type constants and the application of one type to another. Among the usual constants are those for building function types and pairs of types. Type schemes are obtained by quantifying types by type variables as in the scheme for `map` above.

It might be tempting to augment the expressive power of the type system by introducing λ -abstraction with respect to type variables but then β -equality of types destroys the boundary between shape and data. For example, consider the type declaration

```
type Nested  $GFX = \text{Nest of } G(FX)$ 
```

which creates nested data types. If `Nested` is represented by $\Lambda G, F, X. G(FX)$ then `Nested GFX` and `$G(FX)$` are β -equal. However, mapping across the former employs a function of X while the latter employs a function of FX . Hence identifying these types would make it impossible to define generic mapping.

Instead, the necessary expressive power is achieved by introducing type constants in the style of *combinatory algebra* (see e.g. [HS86]). Its two fundamental combinators are K and S whose defining equations are

$$\begin{aligned} KXY &= X \\ SGFX &= GX(FX). \end{aligned}$$

Now K can be modeled directly by a type declaration

$$\text{type } KXY = \text{Evr of } X$$

where the constructor $\text{Evr} : X \rightarrow KXY$ (pronounced “ever”) stands in place of the usual equality. The equation for S does not translate quite so directly, since in the type $GX(FX)$ the data parameter X appears outside the final argument of the application. This is handled by replacing $GX(FX)$ by $G(X, FX)$ in the declaration

$$\text{type } S_1GFX = \text{Rep of } G(X, FX).$$

In pure combinatory algebra all combinators can be generated from S and K . For example, the *identity combinator* I is defined to be SKK since $SKKX = KX(KX) = X$. However, the need to isolate the data parameters means that the combinatory type system cannot be quite so economical. For example, mapping a function from X to Y over $KX(KX)$ would produce a term of type $KX(KY)$ not $KY(KY)$. Hence the identity type must be declared by

$$\text{type } IX = \text{Ths of } X.$$

1.4 Representing data types

Two other basic type constants are

$$\begin{aligned} \text{type } U &= \text{Un} \\ \text{type } BFGX &= \text{Bind of } FX \text{ and } GX. \end{aligned}$$

called the *unit* type and the *parametrised product*. The constants U, K, I and B suffice to represent simple polynomial types underpinning `List` and `Btree0` but another five constants (see Figure 7) are required to represent nested data structures with multiple sorts of data.

These constructors suffice to build concrete representations of arbitrary data structures. The structures themselves are represented by *tagging* concrete representations with *names* using the constructor `Tag`. For example, the *abstractors* `Nil` and `Cons` are defined to be

$$\begin{aligned} \text{Nil} &= \text{Tag nm}(\text{Nil}) (\text{Evr Un}) \\ \text{Cons } x \ y &= \text{Tag nm}(\text{Cons}) (\text{Bind } (\text{Ths } x) \ y). \end{aligned} \tag{1}$$

Then, since the choice of name is irrelevant to generic functions, a single pattern for `Tag` can be used to match against values of arbitrary data type. For example, the case for `map f` on tagged terms is given by

$$| \text{Tag } n \ t \rightarrow \text{Tag } n \ (\text{map } f \ t).$$

Under this interpretation `Cons` is no longer a constructor and so the class of patterns must be expanded to admit `Cons x y` as a pattern which *simplifies* to `Tag nm(Cons) (Bind (Ths x) y)` as described in Section 9.

1.5 Contents of the paper

Section 1 introduces the paper. Section 2 introduces the combinatory types. Section 3 introduces the pattern calculus with its term formation and reduction rules. Section 4 defines generic equality and addition. Section 5 establishes some basic properties of reduction such as the Church-Rosser property and subject reduction. Section 6 defines a type inference algorithm and shows that it is correct. Section 7 introduces type and data type declarations. Section 8 introduces the representing types and generic mapping, and uses them to represent arbitrary data types. Section 9 expands the collection of patterns to allow some pattern simplification. Section 10 defines generic functions for folding and zipping. Section 11 looks at the relationship of this system to system **F** and to other approaches to generic programming. Section 12 looks briefly at future work. Section 13 draws conclusions.

2 Combinatory types

Type contexts (Δ)	$\frac{}{\vdash}$	$\frac{\Delta \vdash}{\Delta, X \vdash} X \notin \Delta$
Types (S, T)	$\frac{}{\Delta \vdash X} X \in \Delta$	$\frac{\Delta \vdash C}{\Delta \vdash C}$ $\frac{\Delta \vdash S \quad \Delta \vdash T}{\Delta \vdash ST}$
Type Schemes (τ)	$\frac{\Delta \vdash T}{\Delta \vdash_s T}$	$\frac{\Delta, X \vdash_s \tau}{\Delta \vdash_s \forall X. \tau}$

Figure 1: Combinatory types

The *combinatory type system* is given in Figure 1. The types are given by a simple combinatory algebra whose expressive power is determined by the choice of constants. Type schemes are created from types by quantifying over type variables.

A *type context* (meta-variable Δ) is a sequence X_1, \dots, X_n of distinct *type variables* (meta-variables X and Y). The judgement $\Delta \vdash$ asserts that Δ is a *well-formed type context*. A *type* (meta-variables S and T) is either a type variable, a *type constant* (meta-variable C) or an *application* ST of a (higher) type S to a type T . Application associates to the left. The judgement $\Delta \vdash T$ asserts that T is a *well-formed type* in type context Δ . The *raw type schemes* (meta-variable τ) are either types or given by quantifying a type scheme by a type variable. The judgement $\Delta \vdash_s \tau$ asserts that τ is a *well-formed raw type scheme* in type context Δ . 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 the type context X_1, \dots, X_n and τ is a type scheme we may write $\forall \Delta. \tau$ in place of $\forall X_1. \forall X_2 \dots \forall X_n. \tau$.

The choice of type constants determines the character of the type system. We will require a constant **Function** where **Function** ST or $S \rightarrow T$ is the type

of *functions* from S to T . For example, by declaring constants for a unit type, binary products, and binary sums we can create the polynomial types familiar in treatments of the Hindley-Milner type system [Mil78]. Some novel type constants will be used to represent data types (Section 8) or to support particular generic functions (Section 10).

In combinatory logic, it is common to omit spaces between the combinators appearing in application. To avoid confusion, let us adopt the convention that actual type constants will either be denoted by a single, upper case italic letter, such as K or as a capitalised word in sans serif, such as `List`.

The presence of unnatural types such as `List List` does not interfere with the developments that follow but if desired they can be excluded by introducing a system of kinds to classify the types as in [Jay01].

A *type substitution* σ is a partial function of finite domain from type variables to types. The action of a substitution σ extends homomorphically to any expression containing type variables (including those to be defined later) but using α -conversion of type schemes to avoid variable capture. The *composition* $\sigma_2\sigma_1$ of two substitutions is defined by $(\sigma_2\sigma_1)X = \sigma_2(\sigma_1X)$. A substitution *from* type context Δ_1 *to* type context Δ_2 is a type substitution σ such that each variable X in the domain of σ is also in Δ_1 and the free variables of σX are all in Δ_2 .

Now let us consider type unification. Some care will be required when typing extensions to ensure that the inferred type of a specialisation is sufficiently general. This is achieved by *fixing* some variables. Let S and T be two types that are well-formed in type context Δ_1, Δ_2 . A *unifier* for them which *fixes* Δ_2 is a substitution $\sigma : \Delta_1 \rightarrow \Delta_3$ such that $\sigma S = \sigma T$. A *most general unifier* for them which *fixes* Δ_2 is a unifier v that fixes Δ_2 such that any other unifier σ for them that fixes Δ_2 factors through v by some substitution ρ so that $\sigma = \rho v$. When Δ_2 is the empty context then σ is a *unifier* for S and T and v is their *most general unifier*.

Lemma 2.1 *If two types S and T that are well-formed in type context Δ_1, Δ_2 have a unifier that fixes Δ_2 then they have a most general unifier that fixes Δ_2 .*

Proof When Δ_2 is empty the result is the standard one due to Robinson [Rob65]. The general result can be achieved by treating the variables in Δ_2 as if they were new constants of the type system and proceeding as before. \square

The unifier produced in the proof above may be denoted $\mathcal{U}(\Delta_2, S, T)$. It has the property of not introducing any fresh type variables so that $\mathcal{U}(\Delta_2, S, T) : \Delta_1\Delta_2 \rightarrow \Delta_1\Delta_2$. If Δ_2 is empty then $\mathcal{U}(\Delta_2, S, T)$ may be written $\mathcal{U}(S, T)$ while $\mathcal{U}(S, T) \uparrow$ indicates that S and T do not have any unifiers.

3 The pattern calculus

3.1 Terms

The term contexts, patterns and terms of the pattern calculus are given in Figure 2. A *term context* (meta-variable Γ) is a finite sequence of distinct *term variables* (meta-variables x and y) with given type schemes. A *context* $\Delta; \Gamma$ is given by a type context Δ and a term context Γ .

Term contexts (Γ)	$\frac{\Delta \vdash}{\Delta; \vdash}$	$\frac{\Delta; \Gamma \vdash \quad \Delta \vdash_s \tau \quad x \notin \Gamma}{\Delta; \Gamma, x : \tau \vdash}$
Patterns (p)	$\frac{}{X; x : X \vdash_o x : X}$	$\frac{}{\Delta; \vdash_o c : T} \quad c : \forall \Delta. T$
Terms (s, t)	$\frac{\Delta; \Gamma \vdash_o p : T \rightarrow S \quad \Delta_1; \Gamma_1 \vdash_o p_1 : T_1 \quad \Delta, \Delta_1; \Gamma, \Gamma_1 \vdash}{\Delta, \Delta_1; v\Gamma_1, v\Gamma \vdash_o p p_1 : vS} \quad v = \mathcal{U}(T_1, T)$	
	$\frac{\Delta; \Gamma \vdash}{\Delta; \Gamma \vdash x : \sigma T}$	$\frac{\Gamma(x) = \forall \Delta_1. T \quad \sigma : \Delta_1 \rightarrow \Delta}{\Delta; \Gamma \vdash c : \sigma T} \quad c : \forall \Delta_1. T$
	$\frac{\Delta; \Gamma \vdash s : T \rightarrow S \quad \Delta; \Gamma \vdash t : T}{\Delta; \Gamma \vdash s t : S}$	
	$\frac{\Delta; \Gamma \vdash t : T \rightarrow S \quad \Delta_1; \Gamma_1 \vdash_o p : T_1 \quad \Delta, \Delta_1; v\Gamma, v\Gamma_1 \vdash s : vS}{\Delta; \Gamma \vdash \text{at } p \text{ use } s \text{ else } t : T \rightarrow S} \quad v = \mathcal{U}(T_1, T)$	
	$\frac{\Delta; \Gamma \vdash t : T \rightarrow S \quad \Delta_1; \Gamma_1 \vdash_o p : T_1 \quad \Delta, \Delta_1; \Gamma, \Gamma_1 \vdash}{\Delta; \Gamma \vdash \text{at } p \text{ use } s \text{ else } t : T \rightarrow S} \quad \mathcal{U}(T_1, T) \uparrow$	
	$\frac{\Delta, \Delta_1; \Gamma \vdash s : S \quad \Delta; \Gamma, x : \forall \Delta_1. S \vdash t : T}{\Delta; \Gamma \vdash \text{let } x = s \text{ in } t : T}$	
	$\frac{\Delta, \Delta_1; \Gamma, x : \forall \Delta_1. T \vdash t : T}{\Delta, \Delta_1; \Gamma \vdash \text{fix } (x, t) : T}$	

Figure 2: The pattern calculus

The *patterns* (meta-variable p) are either term variables, *constructors* (meta-variable c) or *applications* of one pattern to another. The judgement $\Delta; \Gamma \vdash_o p : T$ asserts that p is a pattern of type T in context $\Delta; \Gamma$. Each term variable x is a pattern of variable type. Each constructor c comes equipped with a given closed type scheme $\forall \Delta. T$ and is a pattern of type T . The application of one pattern p to another p_1 requires that their contexts be independent from each other, and that the type of p_1 can be unified with the argument type of p . The type derivation rules for patterns have been arranged to ensure that their types are as general as possible, so that pattern-matching is type-safe. Also, each variable appearing in the term context appears exactly once in the pattern to ensure correct bindings of variables in specialisations.

The *raw terms* (meta-variables s and t) are built from variables, constructors, applications, extensions, let-terms and recursion. The judgement $\Delta; \Gamma \vdash t : T$ asserts that t is a raw term of type T in the context $\Delta; \Gamma$. Let us consider the cases.

If $\Gamma(x) = \forall \Delta_1. T$ then x has type σT for any substitution σ . If $c : \forall \Delta_1. T$ is a constructor then it has type σT for any substitution σ . If $s : T \rightarrow S$ and $t : T$

$$\begin{aligned}
fv(x) &= \{x\} \\
fv(c) &= \{\} \\
fv(s\ t) &= fv(s) \cup fv(t) \\
fv(\text{at } p \text{ use } s \text{ else } t) &= fv(t) \cup (fv(s) - fv(p)) \\
fv(\text{let } x = s \text{ in } t) &= fv(s) \cup (fv(t) - \{x\}) \\
fv(\text{fix}(x, t)) &= fv(t) - \{x\}.
\end{aligned}$$

Figure 3: Free variables

then the *application* $s\ t$ has type S .

The term $\text{at } p \text{ use } s \text{ else } t$ is an *extension* of the *default function* t at the pattern p by the *specialisation* s . When applied to a term t_1 that matches p then s is used else t is used. Any type $T \rightarrow S$ for the whole extension must be a type for the default function. The pattern must have a type T_1 . If T_1 and T have a unifier v then it suffices to type the specialisation by vS in the context obtained by applying v to the combined context created from the previous two premises. Note that the existence of this combined context implies that its components employ distinct type and term variables. The former ensures that the type of the pattern is as general as possible and the latter ensures that the free variables in the pattern p may appear in the specialisation but not in the default. Note that v does *not* appear in the conclusion, which will constrain type inference. If T_1 and T do not have a unifier then specialisation can never occur and so typing is safe. Though distracting in practice (since such failures are usually a sign of programmer error) this rule is necessary to ensure that type derivations are stable under type substitutions.

The term $\text{let } x = s \text{ in } t$ binds x to s in t . When typing t the type scheme for x may bind type variables not appearing in the rest of the term context.

The term $\text{fix}(x, t)$ is a *polymorphic recursion* with recursion variable x or the *fixpoint of t with respect to x* . It takes type T if t has type T in a context where x has type scheme $\forall \Delta_1. T$ where Δ_1 consists of type variables that are not free in the term context. Different uses of the recursion variable x in defining a generic function may exploit different instantiations of its type scheme.

The set of *free variables* $fv(t)$ of a raw term t are defined in Figure 3. The only variables which are not free, i.e. *bound*, are those occurring free in a pattern, let-bound variables or recursion variables. α -conversion of bound variables is defined as usual. A term is *closed* if it has no free variables. *Term substitutions* and their application are then defined in the usual way. The syntax $t\{s/x\}$ denotes the result of substituting s for free occurrences of x in t . The actual *terms* are equivalence classes of raw terms under α -conversion of bound variables.

3.2 Syntactic sugar

The syntax $| p \rightarrow t$ corresponds to the program fragment $\text{at } p \text{ use } t \text{ else}$. A sequence of such fragments produces a pattern-matching program whose ultimate default (if no pattern matches) is an error term `error`. The simplest form of error is non-terminating term of polymorphic type, e.g. $\text{fix}(x.x)$. In FISH2 such

$(\text{at } x \text{ use } s \text{ else } t) t_1$	$>$	$s\{t_1/x\}$
$(\text{at } c \text{ use } s \text{ else } t) c$	$>$	s
$(\text{at } c \text{ use } s \text{ else } t) t_1$	$>$	t if t_1 cannot become c
$(\text{at } p_1 p_2 \text{ use } s \text{ else } t) (t_1 t_2)$	$>$	$(\text{at } p_1$ $\text{use at } p_2 \text{ use } s \text{ else } \lambda y.t (p_1 y)$ $\text{else } \lambda x,y.t (x y))$ $t_1 t_2$ if t_1 is a constructed term
$(\text{at } p_1 p_2 \text{ use } s \text{ else } t) t_1$	$>$	t if t_1 cannot become applicative
$\text{let } x = s \text{ in } t$	$>$	$t\{s/x\}$
$\text{fix } (x, t)$	$>$	$t\{\text{fix } (x, t)/x\}$

Figure 4: Reduction rules for the pattern calculus

errors are defined using an exception constructor so that exception handling can be done by pattern-matching.

A complete term of the form $| p \rightarrow t$ can be regarded as a λ -abstraction $\lambda p.t$ of t with respect to the pattern p (see e.g. [For02]). Also, define $g.f$ to be $\lambda x.g (f x)$ for some fresh variable x . The wildcard symbol $_$ in $| _ \rightarrow t$ represents a fresh variable. We may also write $\text{match } t_1 \text{ with } t$ for $t t_1$ especially when t is given by pattern-matching.

3.3 Reduction

The reduction rules for extensions determine when to specialise and when to default. Some care is required to avoid a default when further reduction could allow specialisation to occur. A *constructed term* is a term t 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. In the latter case t is an *applicative term*. Let c be a constructor. A term t *cannot become* c if it is either an extension, an applicative term or a constant other than c . A term *cannot become applicative* if it is either an extension or a constructor.

The *basic reduction rules* of the pattern calculus are given by the relation $>$ in Figure 4. Let us consider the cases. When the pattern is a variable x then specialisation always occurs with the argument t_1 being substituted for x in the specialisation, just as in β -reduction of λ -abstractions. When the pattern is a constructor c then if the argument is also c then the specialisation is returned. Conversely, if the argument cannot become c then the default is applied to the argument. When the pattern is an application $p_1 p_2$ and the argument is an applicative term $t_1 t_2$ then specialisation occurs. The specialisation tries to match p_1 with t_1 and p_2 with t_2 with failure at any point applying the default to a reconstructed version of $t_1 t_2$. Let-terms are evaluated by substituting for the bound variable. A fixpoint reduces to its body with the recursion variable replaced by the fixpoint.

A *one-step reduction* $t \rightarrow_1 t'$ is given by the application of a basic reduction to a sub-term of t . A *reduction* $t \rightarrow t'$ is given by a finite sequence of one-step reductions $t \rightarrow_1 t_1 \rightarrow \dots \rightarrow t'$.

Unlike most approaches to generic programming, reduction here does not

```

general_equal : X → Y → Bool =
  | x1 x2 → ( | y1 y2 → general_equal x1 y1 && (general_equal x2 y2))
  | x → | y → primequal x y

equal : X → X → Bool = general_equal

```

Figure 5: Equality

require explicit type information to guide specialisation of generic functions. To be sure, constructors always carry some partial, implicit type information but this is not used to drive the evaluation.

The side conditions to the default reduction rules will typically disappear when an evaluation strategy is employed, since reduction of the extension argument to a value means that the default evaluation rules apply exactly when the specialisation rules do not.

4 Basic examples

4.1 Booleans

Booleans are declared by

```
type Bool = True | False.
```

The standard conditional is defined by

```
if b then s else t = match b with | True → s | False → t
```

or (at True use s else at False use t else error) b . Other variations are possible, e.g. by deleting either case, or inserting an ultimate default (e.g. the command that does nothing). The infix operation of conjunction $\&\&$ and other boolean operations, such as $\text{not} : \text{Bool} \rightarrow \text{Bool}$ can be defined in the usual way. It is useful to have a primitive equality

```
primequal : X → Y → Bool
```

to determine equality of constructors. Given a fixed set of constructors, it is defined by pattern-matching against each of them in turn.

The generic equality function is defined in Figure 5. `general_equal` takes two arguments of possibly different types and compares them. The types may be different since the components of distinct terms $x_1 x_2$ and $y_1 y_2$ may have different types even though the applications themselves share a type. The standard typing is imposed afterwards when defining `equal`.

For example, if `Nil` and `Cons` are considered to be constructors then the following evaluation can be performed.

```

general_equal (Cons True Nil) (Cons True Nil) →
general_equal (Cons True) (Cons True) && (general_equal Nil Nil) →
general_equal Cons Cons && (general_equal True True) → True

```

Note how `general_equal` is applied to the partially applied constructor `Cons True`.

4.2 Datum types

It is convenient to introduce some *datum types* that represent atoms of data, here taken to be integers, floating point reals or characters of types `plnt`, `pFloat` and `pChar`, respectively. These types come equipped with some constructors, called *datum constructors* to represent their values (e.g. primitive integers) and some operators, such as the primitive addition of integers $+_i$ and floats $+$. which are used to form terms, e.g. if t_1 and t_2 are both primitive integers then so is $t_1 +_i t_2$. An alternative approach would treat these operators as constants of the language which are not constructors but then the default reduction rule for applications would require modification.

In order to have patterns that match against arbitrary integers it is simplest to embed the primitive integers within an abstract type. More generally, the declarations

```
type Int = Int of plnt
type Float = Float of pFloat
type Char = Char of pChar
```

represent the ordinary integers, floats and characters, respectively. Operations can then be lifted from the primitive integers to the ordinary ones by pattern-matching.

More generally, such operations can be made generic. For example, generic addition is defined in Figure 6. As with `general_equal`, it cannot be guaranteed that the two arguments have the same type, and so `primequal` is used to compare corresponding constructors. For the sake of clarity, its patterns are expressed as pairs given by the declaration

```
type P(X, Y) = Pair of X and Y.
```

We may write $X * Y$ for $P(X, Y)$ and (x, y) for `Pair x y`. More generally still, `general_plus` can be defined as the application of the generic binary operation

```
binaryOp : (plnt → plnt → plnt) → (pFloat → pFloat → pFloat) → X → Y → X
```

to $\lambda x, y. x +_i y$ and $\lambda x, y. x + . y$. Similarly, relations such as less-than can be defined as an application of a generic binary relation.

```
(general_plus : X → Y → X) u v =
  match (u, v) with
  | (Int x, Int y) → Int (x +i y)
  | (Float x, Float y) → Float (x + .y)
  | (x1 x2, y1 y2) → (general_plus x1 y1) (general_plus x2 y2)
  | (x, y) → match primequal x y with | True → x

(plus : X → X → X) = general_plus
```

Figure 6: Addition

5 Properties of reduction

Theorem 5.1 ((Church-Rosser)) *Reduction is Church-Rosser.*

Proof Given a fixed set of constructors, the rules for specialisation can be expanded to a set of rules which are left-linear, as well as non-overlapping (see, for example, [Klo80]). \square

A term is *reducible* if there is a reduction rule that can be applied to one of its sub-terms. Otherwise it is *in normal form* or *normal*. Assume that closed terms built using an operator are always reducible.

Theorem 5.2 *Every closed normal term is either a constructed term or an extension. Hence, the application of an extension to a closed normal term can always be reduced.*

Proof The second assertion follows directly from the first. The first assertion is proved by induction on the structure of a closed normal term t . If t is a constructed term or extension then the result holds. Let-terms, fixpoints and operators applied to closed terms are always reducible and so cannot appear. It remains to consider an application $t_1 t_2$. By induction t_1 is either a constructed term or an extension but it cannot be an extension by the second assertion and so $t_1 t_2$ is a constructed term. \square

Lemma 5.3 *Typings of terms are stable under term substitution. That is, if there are derivations $\Delta; \Gamma, x : S \vdash t : T$ and $\Delta; \Gamma \vdash s : S$ then there is a derivation of $\Delta; \Gamma \vdash t\{s/x\} : T$.*

Proof The proof is by induction on the structure of the derivation of the typing of t . All of the cases are straightforward. \square

Theorem 5.4 ((Subject Reduction)) *Reduction preserves typing.*

Proof The proof is by case analysis on the basic reductions since the result extends to arbitrary reductions by Lemma 5.3. The only interesting cases are the specialisations of extensions as described in Figure 4.

Consider a specialisation at a variable. A type derivation for the left-hand side is given by a derivation

$$\frac{\Delta; \Gamma \vdash t : T \rightarrow S \quad X; x : X \vdash_{\circ} x : X \quad \Delta, X; \Gamma, x : T \vdash s : S}{\Delta; \Gamma \vdash \text{at } x \text{ use } s \text{ else } t : T \rightarrow S}$$

and a derivation $\Delta; \Gamma \vdash t_1 : T$ to produce $\Delta; \Gamma \vdash (\text{at } x \text{ use } s \text{ else } t) t_1 : S$. Now X is not free in Γ or $T \rightarrow S$ and so there is a derivation of $\Delta; \Gamma, x : T \vdash s : S$. Now apply Lemma 5.3 to get the required typing of $s\{t_1/x\}$.

Now consider a specialisation at a constructor. A derivation for the extension must take the form

$$\frac{\Delta; \Gamma \vdash t : T \rightarrow S \quad \Delta_1; \vdash_{\circ} c : T_1 \quad \Delta, \Delta_1; v\Gamma \vdash s : vS}{\Delta; \Gamma \vdash \text{at } c \text{ use } s \text{ else } t : T \rightarrow S} \quad v = \mathcal{U}(T_1, T)$$

and the typing of its application to c requires that $\sigma T_1 = T$ for some substitution σ . Hence σ factors through v and so $v\Gamma = \Gamma$ and $vS = S$ whence $\Delta; \Gamma \vdash s : S$ as required.

Finally consider a specialisation at an application $p_1 p_2$ with type derivation

$$\frac{\Delta_1; \Gamma_1 \vdash_{\circ} p_1 : T_3 \rightarrow T_1 \quad \Delta_2; \Gamma_2 \vdash_{\circ} p_2 : T_4}{\Delta_1, \Delta_2; v_1(\Gamma_1, \Gamma_2) \vdash_{\circ} p_1 p_2 : v_1 T_1} v_1 = \mathcal{U}(T_4, T_3).$$

Then the extension must have a type derivation

$$\frac{\begin{array}{c} \Delta; \Gamma \vdash t : T \rightarrow S \\ \Delta_1, \Delta_2; v_1(\Gamma_1, \Gamma_2) \vdash_{\circ} p_1 p_2 : v_1 T_1 \\ \Delta, \Delta_1, \Delta_2; v v_1(\Gamma, \Gamma_1, \Gamma_2) \vdash s : v S \end{array}}{\Delta; \Gamma \vdash \text{at } p_1 p_2 \text{ use } s \text{ else } t : T \rightarrow S} v = \mathcal{U}(v_1 T_1, T)$$

whose application to $t_1 t_2$ requires derivations of $\Delta; \Gamma \vdash t_1 : T_5 \rightarrow T$ and $\Delta; \Gamma \vdash t_2 : T_5$ for some type T_5 . From these we can create a derivation of $\Delta; \Gamma \vdash \lambda x, y. t (x y) : (T_5 \rightarrow T) \rightarrow T_5 \rightarrow S$. Now consider the extension of this function at p_1 . If $v_2 = \mathcal{U}(T_3 \rightarrow T_1, T_5 \rightarrow T)$ does not exist then we are done, so assume that it does exist so that it suffices to derive $\Delta, \Delta_1; v_2(\Gamma, \Gamma_1) \vdash$ at p_2 use s else $\lambda y. t (p_1 y) : v_2(T_5 \rightarrow S)$. The derivation for the default is straightforward. For the specialisation, assume that the most general unifier $v_3 = \mathcal{U}(T_4, v_2 T_5)$ exists. Then it suffices to derive of $\Delta, \Delta_1, \Delta_2; v_3 v_2(\Gamma, \Gamma_1, \Gamma_2) \vdash s : v_3 v_2 S$. Now $v v_1$ is the most general unifier of the pairs (T_3, T_4) and (T_1, T) . Also, $v_3 v_2 T_3 = v_3 v_2 T_5 = v_3 v_2 T_4$ and $v_3 v_2 T_1 = v_3 v_2 T$. Hence, $v_3 v_2$ factors as $\rho v v_1$ for some substitution ρ . Applying ρ to the previous derivation for s achieves the result. \square

6 Type inference

The type inference algorithm for the pattern calculus is derived from Milner's algorithm \mathcal{W} [Mil78]. Type inference for patterns is straightforward. That for terms is more subtle since extensions (like polymorphic recursions, see e.g. [Hen93]) do not always have principal type schemes.

For example, the term $\lambda x. \text{Un } x \rightarrow \text{Un}$ has type schemes $\forall X. X \rightarrow X$ and also $\forall X. X \rightarrow U$ which do not admit a common generalisation. De-sugaring this example produces $\text{at Un use Un else error}$ which might suggest that `error` is the source of difficulties. However, the recursion-free term

`at Ths use $\lambda x. \text{Ths}$ (not x) else $\lambda f, x. f x$`

is typed by both $(\text{Bool} \rightarrow FY) \rightarrow \text{Bool} \rightarrow FY$ and $(X \rightarrow F\text{Bool}) \rightarrow X \rightarrow F\text{Bool}$ but not their common generalisation $(X \rightarrow FY) \rightarrow X \rightarrow FY$ since any type for this term must contain a reference to `Bool`.

The simplest solution is to require that programmers equip each polymorphic recursion and extension with its exact type scheme $\forall \Delta. T$ (or simply T on the understanding that Δ is as large as possible). In practice, generic programs are created by polymorphic recursion over a pattern-matching for which it suffices to supply merely the overall type of the program, as in the definition of `general_plus` in Figure 6.

Lemma 6.1 *Typings of terms are stable under type substitution. That is, if there is a derivation $\Delta; \Gamma \vdash t : T$ and $\sigma : \Delta \rightarrow \Delta'$ is a substitution then there is a derivation $\Delta'; \sigma \Gamma \vdash t : \sigma T$.*

Proof The proof is by induction on the structure of the derivation. All but one of the cases are completely straightforward. Consider the derivation of an extension where the most general unifier exists as in Figure 2. By induction there is a derivation of $\Delta'; \sigma\Gamma \vdash t : \sigma(T \rightarrow S)$ and $\Delta_1; \Gamma_1 \vdash_\circ p : T_1$ is derived as before. Without loss of generality Δ', Δ_1 is well-formed and $\mathcal{U}(T_1, \sigma T)$ exists and is some substitution v_1 . Now $\sigma T_1 = T_1$ and so $v_1\sigma$ is a unifier of T_1 and T and so factors as ρv for some substitution ρ . Applying ρ to the derivation of s yields $\Delta, \Delta_1; v_1\sigma\Gamma, v_1\Gamma_1 \vdash s : v_1\sigma S$ from which the desired derivation follows. \square

Note that the typings of patterns are *not* stable under type substitutions.

Algorithm \mathcal{W} takes a well-formed context $\Delta_a, \Delta_b; \Gamma$ and a term t whose free variables are all in Γ and a type T such that $\Delta_a, \Delta_b \vdash T$. If successful it returns a type substitution $\mathcal{W}(\Delta_a, \Delta_b, \Gamma, t, T) : \Delta_a \rightarrow \Delta', \Delta_b$ say σ such that $\Delta', \Delta_b; \sigma\Gamma \vdash t : \sigma T$. That is, it produces a typing for t using a substitution σ which fixes Δ_b . To infer a type for a closed term t simply compute $\mathcal{W}(X, \ , \ , t, X)$ (the second and third arguments are empty contexts). The algorithm is defined by induction on the structure of t . In each case σ will denote the desired substitution. The algorithm fails if any step within it fails, e.g. if a unifier does not exist. Consider the cases.

1. t is a variable x where $\Gamma(x) = \forall\Delta_1.T_1$. Then σ is $\mathcal{U}(\Delta_b, T_1, T)$.
2. t is a constant $c : \forall\Delta_1.T_1$. Then σ is $\mathcal{U}(\Delta_b, T_1, T)$.
3. t is an application $t_2 t_1$. Let $\sigma_1 : \Delta_a, X \rightarrow \Delta_1\Delta_b$ be $\mathcal{W}((\Delta_a, X), \Delta_b, \Gamma, t_1, X)$ and $\sigma_2 : \Delta_1 \rightarrow \Delta_2, \Delta_b$ be $\mathcal{W}(\Delta_1, \Delta_b, \sigma_1\Gamma, t_2, \sigma_1(X \rightarrow T))$ where X is a fresh type variable. Then σ is $\sigma_2\sigma_1$.
4. t is at p use s else t . Let v_1 be $\mathcal{U}(\Delta_b, Y_1 \rightarrow Y_2, T)$ where Y_1 and Y_2 are fresh variables. Let $\sigma_2 : \Delta_a, Y_1, Y_2 \rightarrow \Delta', \Delta_b$ be $\mathcal{W}((\Delta_a, Y_1, Y_2), \Delta_b; v_1\Gamma, t, v_1T)$ and define σ to be $\sigma_2 v_1$. Let $\Delta_1; \Gamma_1 \vdash_\circ p : T_1$ be a typing for p . The construction of this typing (if it exists) is unique up to the choice of variable names, as is easily established by induction on the structure of p . Let v_3 be $\mathcal{U}(\Delta_b, T_1, \sigma Y_1)$. If $\mathcal{W}(\ , (\Delta', \Delta_b, \Delta_1), v_3\sigma(\Gamma, \Gamma_1), s, v_3\sigma Y_2)$ succeeds (i.e. while fixing all variables) then the algorithm succeeds and returns σ . Note that if further substitution were required to type s then it would not be clear how to separate its effect from that of v_3 .
5. t is let $x = t_1$ in t_2 . Let Y_1 be a fresh type variable and let $\sigma_1 : \Delta_a, Y_1 \rightarrow \Delta_1, \Delta_2$ be $\mathcal{W}((\Delta_a, Y_1), \Delta_b; \Gamma, t_1, Y_1)$ where Δ_1 consists of the type variables free in $\sigma_1\Gamma$. Let $\sigma_2 : \Delta_1 \rightarrow \Delta', \Delta_b$ be $\mathcal{W}(\Delta_1, \Delta_b, \sigma_1\Gamma, x : \forall\Delta_2.\sigma_1 Y_1, t_2, \sigma_1 T)$. Then $\sigma = \sigma_2\sigma_1$.
6. t is $\text{fix}(x.t')$. Decompose Δ_a as Δ_1, Δ_2 where Δ_2 contains those variables in Δ_a which are free in Γ . If $\mathcal{W}(\ , \Delta; \Gamma, x : \forall\Delta_2.T, t', T)$ succeeds (i.e. while fixing all variables in Δ) then σ is the identity substitution.

Theorem 6.2 *Type inference is correct: if $\mathcal{W}(\Delta_a, \Delta_b, \Gamma, t, T)$ succeeds and is $\sigma : \Delta_a \rightarrow \Delta', \Delta_b$ then $\Delta', \Delta_b; \sigma\Gamma \vdash t : \sigma T$.*

Proof The proof is by straightforward induction on the structure of t . \square

Evidence for the usefulness of this algorithm is that it is able to type all of the examples in the paper with the support of the type information given.

7 Declared types

The expressive power of the pattern calculus is highly dependent on the choice of type constants and constructors. This section introduces machinery for declaring types and constructors. The declarations allow recursion on *all* type parameters, whether “higher-order” or not. Also, the treatment of type constructors as types in their own right automatically includes the “nested data types” in the sense of [BM98] (which allow, but do not require, any nesting). The definitions can be expanded to allow mutually recursive data type definitions if desired.

The following section will show how to represent arbitrary data types and how to define generic functions upon them. To do this the data types must be identified, and also their data (type) parameters upon which such functions act. Some examples will clarify the issues.

The key difference between data types and ordinary types is that function types are never data types. Although function types may be thought to support some form of mapping or addition, none of the generic functions defined in this paper extends to functions in an appropriate way: equality risks ignoring renaming of bound term variables, mapping would only apply to arguments in positive positions (e.g. to Y but not X in $X \rightarrow Y$), folding would require the function to have a known finite domain, and zipping would be ambiguous (there being two distinct functions of type $((X \rightarrow Z) \rightarrow Z) \rightarrow ((Y \rightarrow Z) \rightarrow Z) \rightarrow (X * Y \rightarrow Z) \rightarrow Z$).

The simplest response would be to ban function types from type declarations altogether, but the need to declare types like that of arrows in Section 1.2, and to allow its constants `Onefun` and `Bthfun` as constructors makes this impractical. Hence the data type declarations must be distinguished among type declarations in general.

The second issue concerns the identification of the data parameters. The answer adopted is very simple: to map across something of type FX is to act on data of type X leaving F alone. For example, to map across a parametrised product of type $BF GX$ is to apply a function of X while ignoring F and G . In other words, X will be the data parameter when declaring $BF GX$. The consequence of this approach is that all data upon which generic functions are to act must be collected within the last argument of a type application, for which a new type constant is required.

If S and T are types then `CommaST` or (S, T) is the *pair* of S and T . Define a *tuple of type variables* (meta-variable Z) to be either a type variable or a pair of tuples of types. Tuples of other sorts of types, e.g. data types, are defined similarly.

A *simple type declaration* $\text{dec}(D)$ for a *type constant* D is given by a declaration of the form

$$\begin{array}{l} DZ_1 \dots Z_q = \\ \quad c_1 \text{ of } T_{1,1} \text{ and } \dots \text{ and } T_{1,j_1} \\ \quad | \dots \\ \quad | c_n \text{ of } T_{n,1} \text{ and } \dots \text{ and } T_{n,j_n} \end{array}$$

such that each Z_i for $1 \leq i \leq q$ is a tuple of type variables each of which is distinct from all others in $DZ_1 \dots Z_q$ and each $T_{i,j}$ is a type whose free type variables are all free in $DZ_1 \dots Z_q$. The *arity* of the declaration is q . The variables in Z_i for $1 \leq i \leq q$ are called *parameters* with those in Z_q also called

data parameters. Let Δ be the type context consisting of all the parameters. Each c_i is an *declared constructor* whose type scheme is

$$c_i : \forall \Delta. T_{i,1} \rightarrow \dots \rightarrow T_{i,j_i} \rightarrow DZ_1 \dots Z_q.$$

A *type declaration* for a *declared type constant* D of arity q is a declaration of the form

$$\text{type } D \text{ has } \text{dec}_1(D) \text{ or } \dots \text{ or } \text{dec}_m(D)$$

where each $\text{dec}_i(D)$ (for $1 \leq i \leq m$) is a simple type declaration for D of arity q . If $q = 0$ then D is a *parameter-free* declared type. Otherwise it is a *parametrised* declared type. A trivial example of a parameter-free declared type is the *unit type*

$$\text{type } U = \text{Un}.$$

A *simple data type declaration* is a simple type declaration as above such that each $T_{i,j}$ is a data type. A *data type declaration* is a type declaration such that each of its simple declarations is a simple data type declaration. In this case the declared type is a *declared data type*.

A *data type* is either a type variable or of the form $DT_1 \dots T_q$ where D is a declared data type of arity q and T_q is a tuple of data types.

For example, the declarations of lists and trees in the introduction are simple data type declarations. The declaration of the type A of arrows (in Section 1.2) is an example of a type declaration that contains two simple declarations, the first of which is not a data type declaration since it employs a function type.

The *abstractors* are the declared constructors of the parametrised data type declarations. A *data structure* is a term generated by abstractors, datum values and application. Their interpretation in the pattern calculus will be the subject of the next section. All other declared constructors are simply constructors of the calculus.

8 Representing data structures

This section will show how to represent each data structure t as $\text{Tag } n \ t'$ where t' is its concrete representation, n is a name, and

$$\text{Tag} : (F \rightarrow G) \rightarrow FX \rightarrow GX$$

is a new constructor. The process can be outlined as follows. Every data structure has a concrete representation as a finitely branching tree which can be expressed using a small number of representing constructors (and Un). That is, each abstractor can be described using these constructors together with a name that identifies it among all such having the same concrete structure. The story is not quite trivial since generic operations, e.g. to map several functions over such a tree, require the separation of both data from structure, and different sorts of data which implies that data parameters must be tracked precisely. Section 8.1 introduces the representing constructors. Section 8.2 introduces the generic function map by pattern-matching against the representing constructors and Tag : it is used to map representations over nested structures. Section 8.3 shows how to extract data parameters from the types of constructor arguments. Section 8.4 uses this to represent arbitrary abstractors.

```

type KXY = Evr of X
type IX = Ths of X
type BFGX = Bind of FX and GX
type LF(X, Y) = Asl of FX
type RF(X, Y) = Asr of FY
type S1GFX = Rep of G(X, FX)
type S2GF(X, Y) = Rpp of G(X, (FX, Y))
type MF(W, (X, (Y, Z))) = Mid of F(W, ((X, Y), Z))

```

Figure 7: The representing data types

8.1 The representing types

The *representing types* for the higher data types are declared in Figure 7. The constructor *Evr* for the *konstant* type *K* converts a term of type *X* into a structure holding (no) *Y*s of type *KXY*. For example, the Nil list has concrete form given by

$$\text{Evr Un} : KUX.$$

The constructor *Ths* (pronounced “this”) for the *identity type* *I* converts data of type *X* into a data structure holding an *X*. The constructor *Bind* for the *binding type* *B* converts a pair of data structures into a single data structure. For example, *Cons x y* has concrete form

$$\text{Bind (Ths } x) y : B\text{IList}X$$

The constructors *Asl* (pronounced “as left”) and *Asr* (pronounced “as right”) are for *left type* *L* and *right type* *R*, respectively. They are used to introduce pairing of types. For *Btree* (defined in Section 1.2) the concrete form of *Leaf x* is *Asl (Ths x) : LI(X, Btree(X, Y))* and that of *Node x y z* is

$$\text{Bind (Asr (Ths } x)) (\text{Bind } y z) : B(RI)(B \text{ Btree Btree})(X, Y).$$

The constructors *Rep* and *Rpp* for the types *replicate* *S₁* and *parametrized replicate* *S₂* are used to represent nested data structures. For example, *Nest x : Nested GFX* has concrete form *Rep (Asr x) : S₁(RG)FX*. Similarly, *Rpp* will be used to handle multiple data types. For example, given the declaration

$$\text{type Btree}_3 X = \text{Uniform}_2 \text{ of Btree}(X, X)$$

and the term *map2* from Figure 8 then the concrete representation of *Uniform₂* is

$$\text{Rep.Rpp.Asr.}(\text{map2 Ths Ths}) : \text{Btree}(X, X) \rightarrow S_1(S_2(R \text{ Btree}))IIX.$$

Longer tuples of types can be handled by inserting more copies of *Rpp*. Finally, *Mid* re-orders the arguments within a tuple into a right-associative form. Its use can be avoided by forbidding left-associative tuples in type declarations. As a fresh example, let us consider how to represent *Pair : X → Y → X * Y* in this approach. First *X* is replaced by *IX* and then *LI(X, Y)*. Similarly, *Y* becomes by *RI(X, Y)*. Then binding these together represents *Pair x y* by

$$\text{Bind (Asl (Ths } x)) (\text{Asr (Ths } y)) : B(LI)(RI)(X, Y).$$

8.2 Mapping

```

type A has
  AXY = Onefun X → Y
or A(X1, X2)(Y1, Y2) = Bthfun A(X1, Y1) and A(X2, Y2)

(map : AXY → FX → FY) f =
  | Evr x → Evr x
  | Ths x → (match f with | Onefun f1 → Ths (f1 x))
  | Bind x y → Bind (map f x) (map f y)
  | Asl x → (match f with | Bthfun f1 f2 → Asl (map f1 x))
  | Asr x → (match f with | Bthfun f1 f2 → Asr (map f2 x))
  | Rep x → Rep (map (Bthfun f (Onefun (map f))) x)
  | Rpp x → (match f with
    | Bthfun f1 f2 →
      Rpp (map (Bthfun f1 (Bthfun (Onefun (map f1)) f2)) x))
  | Mid x → (match f with
    | Bthfun f1 (Bthfun f2 (Bthfun f3 f4)) →
      Mid (map (Bthfun f1 (Bthfun (Bthfun f2 f3) f4)) x))
  | Tag n x → Tag n (map f x)

map1 f = map (Onefun f)
map2 f g = map (Bthfun (Onefun f) (Onefun g))

```

Figure 8: Mapping

Generic mapping is defined in Figure 8. For example, mapping a function f across Nil evaluates as follows:

$$\begin{aligned}
 \text{map1 } f \text{ Nil} &= \text{map (Onefun } f) (\text{Tag nm(Nil) (Evr Un)}) \\
 &\rightarrow \text{Tag nm(Nil) (map (Onefun } f) (\text{Evr Un})) \\
 &\rightarrow \text{Tag nm(Nil) (Evr Un)}.
 \end{aligned}$$

The pattern for $\text{Rep} : G(X, FX) \rightarrow S_1 GFX$ is typical of the more complex cases. As $\text{map } f : FX \rightarrow FY$ then $\text{Bthfun } f (\text{Onefun (map } f)) : A(X, FX)(Y, FY)$ which can then be mapped over something of type $G(X, FX)$.

8.3 Extracting data parameters

Let Z be a tuple of distinct type variables. A type F is Z -free if no variable in Z also appears in F . A type T is Z -separated if it is of the form FZ for some Z -free type F .

To each type of the form $G(Z, T)$ where T is a tuple of Z -separated types is associated a *tuple extractor* $e_t[G, Z, T] : G(Z, T) \rightarrow F_t[G, Z, T]Z$ of Z from T in G where $F_t[G, Z, T]$ is a Z -free type. It is defined by induction with respect to the structure of T . If T is an application FZ then it is

$$\text{Rep} : G(Z, FZ) \rightarrow S_1 GFZ.$$

If T is a pair of the form (FZ, T_2) then it is

$$e_t[S_2GF, Z, T_2].\text{Rpp} : G(Z, (FZ, T_2)) \rightarrow S_2GF(Z, T_2) \rightarrow F_t[S_2GF, Z, T_2]Z.$$

If T is a pair of the form $((T_1, T_2), T_3)$ then it is

$$\begin{aligned} e_t[MG, Z, (T_1, (T_2, T_3))].\text{Mid} & : G(Z, ((T_1, T_2), T_3)) \rightarrow MG(Z, (T_1, (T_2, T_3))) \\ & \rightarrow F_t[MG, Z, (T_1, (T_2, T_3))]Z \end{aligned}$$

Given Z as above, each data type T has an *extractor* $e[T, Z] : T \rightarrow F[T, Z]$ of Z from T where $F[T, Z]$ is a Z -free type, defined by induction on the structure of T . If T does not contain any of the free variables of Z then it is $\text{Evr} : T \rightarrow KTZ$. If T is a variable in Z then proceed by induction on the structure of Z . If Z is T then the desired embedding is

$$\text{Ths} : Z \rightarrow IZ.$$

If Z is (Z_1, Z_2) then T is free in one of Z_1 and Z_2 . If the former then it is

$$\text{Asl}.e[T, Z_1] : T \rightarrow F[T, Z_1] \rightarrow LF[T, Z_1]Z.$$

Dually, if T is a variable in Z_2 then the desired embedding is

$$\text{Asr}.e[T, Z_2] : T \rightarrow F[T, Z_2] \rightarrow RF[T, Z_2]Z.$$

If T is an application GT' where T' is a tuple of data types then proceed as follows. Define $f : AT'T''$ by induction on the structure of T' . If it is a data type then f is $\text{Onefun } e[T', Z]$. If it is a pair (T'_1, T'_2) then define f to be $\text{Bthfun } f_1 f_2$ where $f_1 : AT'_1T''_1$ and $f_2 : AT'_2T''_2$ are given by induction. Then T'' is a tuple of Z -separated types and the desired embedding is

$$e_t[D, Z, T'']. \text{Asr}(\text{map } f) : GT' \rightarrow GT'' \rightarrow RG(Z, T'') \rightarrow F_t[RG, Z, T'']Z.$$

8.4 Defining abstractors

Let $a : T_1 \rightarrow \dots \rightarrow T_n \rightarrow DZ_1 \dots Z_{q-1}Z$ be an abstractor. Define a Z -free type $F_i(a)$ and a term $\text{concr}_i(a) : T_i \rightarrow \dots \rightarrow T_n \rightarrow F_i(a)Z$ by

$$\begin{aligned} \text{concr}_n(a) & = e[T_n, Z] : T_n \rightarrow F[T_n, Z] \\ \text{concr}_i(a) x_i \dots x_n & = \text{Bind}(e[T_i, Z] x_i)(\text{concr}_{i+1}(a) x_{i+1} \dots x_n) \\ & : BF[T_i, Z]F_{i+1}(a)Z. \end{aligned}$$

Define $F(a) = F_1(a)$ and $\text{concr}(a) = \text{concr}_1(a) : T_1 \rightarrow \dots \rightarrow T_n \rightarrow F(a)Z$. Now introduce a constructor $\text{nm}(a) : F(a) \rightarrow DZ_1 \dots Z_{p-1}$ called the *name* of a and *define* a to be the term

$$a = \lambda x_1, \dots, x_n. \text{Tag nm}(a) (\text{concr}(a) x_1 \dots x_n).$$

9 Evaluating patterns

The above account of abstractors allows generic functions such as `map` to be applied to arbitrary data structures since they evaluate to tagged forms. The price to be paid is that abstractors are no longer constructors and so, according to Figure 2, programs like `length` in Section 1.1 are not well-formed! The solution is to accept the λ -abstraction `Cons` as a pattern, and admit the reduction of the pattern `Cons h t` to its value `Tag nm(Cons) (Bind (Ths x) y)`. That is, add the type derivation rule

$$\frac{\Delta; \Gamma, x : T_1 \vdash_{\circ} p : T_2}{\Delta; \Gamma \vdash_{\circ} \lambda x.p : T_1 \rightarrow T_2}$$

to the system defined in Figure 2 and the β -reduction rule $(\lambda x.p) p_1 > p\{p_1/x\}$ to basic reductions in Figure 4. This reduction is not as powerful as it may at first appear since each term variable in the context for a pattern must appear exactly once, so evaluation cannot duplicate or eliminate variables. That is, such reduction is merely a form of *simplification*.

Type inference is unaffected by these additions. The proofs of the Church-Rosser property and subject reduction for pattern reduction are straightforward.

One could generalise further and define patterns by extensions and fixpoint constructions, too but some care is required to respect the occurrences of free variables.

The other challenge to the abstractness of abstractors is that pretty printing of results is likely to reveal concrete representations such as `Tag nm(Nil) (Evr Un)` when an abstractor such as `Nil` is to be preferred. This requires a pretty-printer that is able to recover the data from within the concrete representation by reversing the process that created it (using eliminators for the corresponding constructors). This is easily described at the same time that abstractors are given their values but we shall not go into the details here.

10 Second-order examples

Generic mapping has already been defined in Figure 8. The other canonical second-order functions are folding and zipping. The function `foldleft` is defined in Figure 9. Its most common use is in `foldleft1` which, when applied to f, x and a list y_1, \dots, y_n produces $f(\dots(f x y_1)\dots) y_n$. We can use it to define the size of data structure by

$$\text{size} = \text{foldleft1} (\lambda x, y.x + 1) 0 : FY \rightarrow \text{Int}. \quad (2)$$

In general we must consider a higher type F that takes more than one argument. For example, to fold over a type $F(Y_1, Y_2)$ first combine functions $f_i : X \rightarrow Y_i \rightarrow X$ for $i = 1, 2$ into a single structure using `ParamOne` and `ParamBth`. Similarly, one can define `ArrowParam` dually to `ParamArrow` and `foldright : ArrowParamXY \rightarrow FX \rightarrow Y \rightarrow Y` whose only essential difference from `foldleft` is that folding over some `Bind x1 x2` acts on x_2 first and then x_1 .

Now consider zipping as defined in Figure 10. The type `Arrow2` is used to represent tuples of functions. On lists, `zipwith1` takes a function $f : X \rightarrow Y \rightarrow Z$ and a pair of lists, one of X s and one of Y s and produces a list whose entries are given by applying f to corresponding list entries. Both lists must have the same

```

type ParamArrow has
  ParamArrow XY = ParamOne of X → Y → X
or ParamArrow X(Y1, Y2) = ParamBth of ParamArrow XY1 and ParamArrow XY2

(foldleft : ParamArrow XY → X → FY → X) f x u =
match (u, f) with
| (Evr y, -) → x
| (Ths y, ParamOne f1) → f1 x y
| (Bind y1 y2, -) → foldleft f (foldleft f x y1) y2
| (Asl y, ParamBth f1 f2) → foldleft f1 x y
| (Asr y, ParamBth f1 f2) → foldleft f2 x y
| (Rep y, -) → foldleft (ParamBth f (ParamOne (foldleft f))) x y) f
| (Rpp y, ParamBth f1 f2) →
  foldleft (ParamBth f1 (ParamBth (ParamOne (foldleft f1)) f2)) x y
| (Mid y, ParamBth f1 (ParamBth f2 (ParamBth f3 f4))) →
  foldleft (ParamBth f1 (ParamBth (ParamBth f2 f3) f4)) x y
| (Tag n y, -) → foldleft f x y

(foldleft1 : (X → Y → X) → X → FY → X) f = foldleft (ParamOne f)

```

Figure 9: foldleft

length. In general both structures must have the same shape. This comparison requires `general_zipwith`, like `general_equal`, to accept arguments created from different types.

11 Relationship to Other Systems

11.1 System **F**

It is common to consider the type systems underpinning functional programming languages as fragments of System **F** [GLT89] whose types

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

consist of variables, function types and quantified types. However, this interpretation does not apply to the pattern calculus at either a conceptual or a technical level.

Technically, the difficulty is that System **F** does not support most general unifiers of types. In standard functional languages unifiers appear in type inference but here they are essential to the type derivation rules for patterns and extensions.

Conceptually, the two systems have incompatible accounts of data types. One of the chief attractions of System **F** is that data types can be expressed as (quantified) function types. For example, the coproduct type $X + Y$ is just

$$\forall Z.(X \rightarrow Z) \rightarrow (Y \rightarrow Z) \rightarrow Z.$$

```

type Arrow2 has
  Arrow2 XYZ = Onefun2 of X → Y → Z
or Arrow2(X1, X2)(Y1, Y2)(Z1, Z2) =
  Bthfun2 of Arrow2 X1Y1Z1 and Arrow2 X2Y2Z2

(general_zipwith : Arrow2XYZ → FX → GY → FZ) f u v =
match (u, v, f) with
| (Evr x, Evr y, _) → ( | True → Evr x) (general_equal x y)
| (Ths x, Ths y, Onefun2 f1) → Ths (f1 x y)
| (Bind x1 x2, Bind y1 y2, _) →
  Bind (general_zipwith f x1 y1) (general_zipwith f x2 y2)
| (Asl x, Asl y, Bthfun2 f1 f2) → Asl (general_zipwith f1 x y)
| (Asr x, Asr y, Bthfun2 f1 f2) → Asr (general_zipwith f2 x y)
| (Rep x, Rep y, _) →
  Rep (general_zipwith (Bthfun2 (Onefun2 (general_zipwith f))) x y)
| (Rpp x, Rpp y, Bthfun2 f1 f2) →
  Rpp (general_zipwith (Bthfun2 f1 (Bthfun2 (Onefun2
    (general_zipwith f1 f2))) x y)
| (Mid x, Mid y, Bthfun2 f1 (Bthfun2 f2 (Bthfun2 f3 f4))) →
  Mid (general_zipwith (Bthfun2 f1 (Bthfun2 (Bthfun2 f2 f3) f4)) x y)
| (Tag m x, Tag n y, _) →
  match primequal m n with | True → Tag m (general_zipwith f x y)

(zipwith : Arrow2 XYZ → FX → FY → FZ) = general_zipwith
(zipwith1 : (X → Y → Z) → FX → FY → FZ) f = zipwith (Onefun2 f)

```

Figure 10: zipwith

That is, given a value of type $X + Y$ and two functions of types $X \rightarrow Z$ and $Y \rightarrow Z$ one can obtain a Z (by case analysis). By introducing abstraction $\Lambda X.T$ of a type T with respect to a type variable X one can define the coproduct itself as

$$\Lambda X, Y. \forall Z. (X \rightarrow Z) \rightarrow (Y \rightarrow Z) \rightarrow Z.$$

That is, the coproduct is defined by its *behaviour* rather than its *structure*. While extremely elegant, this approach makes it impossible to define generic functions that act on data structures but not on λ -abstractions. Also, as noted in the introduction, abstraction with respect to type variables introduces beta-equality of types which prevents them from expressing the structure of data types.

11.2 Pattern-matching

There is a large literature on pattern-matching as a programming tool but our concern is with its use as a fundamental programming construct, on a par with λ -abstraction. The *typed pattern calculus* of [?] represents pattern-matching using ideas from sequent calculus: a pattern-matching term takes the form $\lambda P.M$ where P is a typed pattern, typically containing several alternatives,

and M is a term containing the results for those alternatives. For example, a pattern-match over the constructors `Nil` and `Cons` might take the form $\lambda(\text{Nil} \mid_{\xi} \text{Cons } x \text{ } xs).[M \mid_{\xi} N]$ where ξ is used to relate `Nil` to M and `Cons` x xs to N . This mechanism is further developed in [?].

11.3 Generic programming

Most other treatments of generic programming either focus on the semantics [MFP91, CF92], or use type information to drive the evaluation [JC94, ACPR95, JJ97, Jan00, Hin00, Hin02]. Most of the latter approaches have been unable to handle either nested data structures or more than two sorts of data. In part this has been a conceptual difficulty, but it has been compounded by the inability of host languages such as ML and Haskell to distinguish different ways of nesting structures.

The latest approach by Hinze [Hin02] is able to handle concrete versions of most of the data types considered here. However, it is not very clear how type declarations are to be handled. Like the combinatory type system, it uses a single algorithm for mapping over all types. However, the type of a mapping is computed from the type of its data structure argument. This requires explicit type information for evaluation as these are integral to the algorithm. In practice, pre-processing of the whole program is used to specialise algorithms for the given types.

By contrast, the pattern calculus and its antecedents are able to support evaluation which is truly type-free and modular. This is most clearly seen in Figure 4. Let us consider the antecedents briefly.

Functorial ML [JBM98] used a type system based on functors with a typical data type having the form $F(X_0, X_1, X_2)$ where the arguments of the functor F form an unstructured sequence. The individual type arguments X_i were accessed by their indices i which then infected the whole language. In the same way, mapping was given by a family map^n of functions which, since they were all interdependent, had to be treated as primitive constants rather than defined by pattern-matching.

The earlier work also used a *functorial type system*. It introduced the pairing of types but limited the application of types to the application of a *functor* to a tuple of types. This system was able to handle common nested structures but its limitations motivated the introduction of the combinatory types. Another interesting change is in the treatment of recursive data types. Initial algebras were given by a constructor

$$\text{intrl} : F(X, (\mu F)X) \rightarrow (\mu F)X$$

where μF is the initial algebra for F . This approach constrains the application of the recursive type μF to a single choice of arguments X and so is not able to model declarations $\text{type } D \text{ } X = \dots$ in which D is applied to types other than X on the right-hand side of the declaration. In the combinatory type system such types are handled using data type definitions, so there is no need for a separate pattern for `intrl`, just as there is no need for patterns to handle coproducts (though cases for coproducts continue to appear in other approaches).

The pattern calculus itself evolved from the *constructor calculus* [Jay01]

which uses a weaker form of extension with the syntax

$$\text{under } c \text{ apply } f \text{ else } g$$

where c is a constructor. If c takes n arguments then this can be translated to $\text{at } c \ x_1 \ \dots \ x_n \ \text{use } f \ x_1 \ \dots \ x_n \ \text{else } g$. The use of patterns rather than constructors confers several advantages: the pattern syntax is more expressive as it allows a number of constructors and variables to be combined within a single pattern; the typing and reduction rules do not need to work with the head c of a term $c \ t_1 \ \dots \ t_n$ but just with applicative terms; and the ability to put a variable at the head of a pattern allows one to define functions like equality or size extremely concisely.

12 Future Work

The focus of this paper has been to demonstrate the expressive power of extensions in a purely functional setting. Successors to this paper will show how to add extra features. For example, adding some primitive imperative features is enough to support the definition of a generic assignment

$$\text{assign} : \forall X. \text{loc } X \rightarrow X \rightarrow \text{comm}$$

where $\text{loc } X$ represents a location for a value of type X and comm is a type of commands. It will perform in-place update when safe to do, and allocate fresh memory otherwise.

Another possible development is of generic functions for distributing data across a parallel (or distributed) systems or for developing generic skeletons (see e.g. [HM99]).

A third direction being explored is a new interpretation of object-orientation: the ability to define functions with different algorithms for different types removes a major obstacle to the integration of functional and object-oriented techniques, and the improvement of type systems for object-orientation.

It is desirable to determine when a set of patterns is complete for a given type. Basically, it is complete for a type variable X representing a data type if it contains a pattern that is a mere variable. It is complete for the application FX of one type variable to another if it contains patterns that cover Tag and all of the constructors for the representing types. This could be elaborated into a general account.

Informally, the declared data types can be modeled as functors between categories (e.g. [BW90]) but the standard categorical semantics typically represent types as objects in a category and terms as arrows. A formal denotational semantics must reconcile these viewpoints, perhaps using some internal category theory to encode functors within a single category.

As mentioned in the introduction, the pattern calculus and the combinatorial types underpin the development of the programming language FISH2. The implementation is robust enough to type all of the examples in this paper and is available on request.

13 Conclusions

The pattern calculus provides a natural and powerful account of pattern-matching which is distinguished by the ability to combine cases that have different types. With the combinatory type system it provides a complete solution to the problem of generic programming in the following precise sense. Arbitrary data structures can be represented by attaching names to concrete representations built with the constants of primitive datum type and the constructors `Un`, `Evr`, `Ths`, `Bind`, `Asl`, `Asr`, `Rep`, `Rpp` and `Mid`. Generic functions for operations such as equality, addition, mapping, folding and zipping can be defined by pattern-matching over these constructors, and can be evaluated using a handful of type-free reduction rules. Standard results such as subject reduction hold. Type inference is practical since programmers are required to supply the types of generic functions at the point of definition but not at the point of use.

Acknowledgements The Université de Paris VII and the University of Edinburgh each hosted me for a month while working on this material, the latter by EPSRC visiting fellowship No. GR/M36694. I would like to thank my hosts Pierre-Louis Curien and Don Sannella for making these visits so pleasant and productive. This work has also profited from discussions with many other colleagues, especially Manuel Chakravarty, John Crossley, Peter Dybjer, Martin Hofmann, Neil Jones, Gabi Keller, Hai Yan Lu, Greg Michaelson, Eugenio Moggi, Jens Palsberg, Tony Nguyen, Gilles Peskine, David Skillicorn, Bob Tennent and Gyun Woo. I would also like to thank the anonymous referees for their careful and constructive criticism.

References

- [ACPR95] M. Abadi, L. Cardelli, B.C. Pierce, and D. Rémy. Dynamic typing in polymorphic languages. *Journal of Functional Programming*, 5(1):111–130, 1995.
- [BM98] Richard Bird and Lambert Meertens. Nested datatypes. In J. Jeuring, editor, *Proceedings 4th Int. Conf. on Mathematics of Program Construction, MPC'98, Marstrand, Sweden, 15–17 June 1998*, volume 1422, pages 52–67. Springer-Verlag, Berlin, 1998.
- [BS98] R. Backhouse and T. Sheard, editors. *Workshop on Generic Programming: Marstrand, Sweden, 18th June, 1998*. Chalmers University of Technology, 1998.
- [BW90] M. Barr and C. Wells. *Category Theory for Computing Science*. International Series in Computer Science. Prentice Hall, 1990.
- [Cam97] OBJECTIVE CAML. <http://pauillac.inria.fr/ocaml>, 1997.
- [CF92] J.R.B. Cockett and T. Fukushima. About Charity. Technical Report 92/480/18, University of Calgary, 1992.
- [For02] Julien Forest. A weak calculus with explicit operators for pattern matching and substitution. In Sophie Tison, editor, *Rewriting Techniques and Applications, 13th International Conference, RTA 2002*,

- Copenhagen, Denmark, July 22-24, 2002, Proceedings*, volume 2378, pages 174–191. Springer, 2002.
- [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.
- [GLT89] J-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*. Tracts in Theoretical Computer Science. Cambridge University Press, 1989.
- [Has02] HASKELL. <http://www.haskell.org/>, 2002.
- [Hen93] F. Henglein. Type inference with polymorphic recursion. *ACM Trans. on Progr. Lang. and Sys.*, 15:253–289, 1993.
- [Hin00] Ralf Hinze. A new approach to generic functional programming. In *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 119–132. ACM Press, 2000.
- [Hin02] Ralf Hinze. Polytypic values possess polykinded types. *Science of Computer Programming*, 43:129–159, 2002.
- [HM99] K. Hammond and G. (eds) Michaelson. *Research Directions in Parallel Functional Programming*. Springer, 1999.
- [HS86] R. Hindley and J.P. Seldin. *Introduction to Combinators and Lambda-calculus*. Cambridge University Press, 1986.
- [Jan00] P. Jansson. *Functional Polytypic Programming*. PhD thesis, Chalmers University, 2000.
- [Jay95] C.B. Jay. A semantics for shape. *Science of Computer Programming*, 25:251–283, 1995.
- [Jay96] C.B. Jay. Shape in computing. *ACM Computing Surveys*, 28(2):355–357, 1996.
- [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.
- [JBM98] C.B. Jay, G. Bellè, and E. Moggi. Functorial ML. *Journal of Functional Programming*, 8(6):573–619, 1998.
- [JC94] C.B. Jay and J.R.B. Cockett. Shapely types and shape polymorphism. In D. Sannella, editor, *Programming Languages and Systems - ESOP '94: 5th European Symposium on Programming, Edinburgh, U.K., April 1994, Proceedings*, Lecture Notes in Computer Science, pages 302–316. Springer Verlag, 1994.

- [Jeu00] J. Jeuring, editor. *Proceedings: Workshop on Generic Programming (WGP 2000): July 6, 2000, Ponte de Lima, Portugal*. Utrecht University, UU-CS-2000-19, 2000.
- [JJ97] P. Jansson and J. Jeuring. PolyP - a polytypic programming language extension. In *POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 470–482. ACM Press, 1997.
- [Klo80] J.W. Klop. *Combinatory Reduction Systems*. PhD thesis, Mathematical Center Amsterdam, 1980. Tracts 129.
- [MFP91] E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In J. Hughes, editor, *Proceeding of the 5th ACM Conference on Functional Programming and Computer Architecture*, volume 523 of *Lecture Notes in Computer Science*, pages 124–44. Springer Verlag, 1991.
- [Mil78] R. Milner. A theory of type polymorphism in programming. *JCSS*, 17, 1978.
- [MT91] R. Milner and M. Tofte. *Commentary on Standard ML*. MIT Press, 1991.
- [Rob65] J.A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the Association for Computing Machinery*, 12:23–41, 1965.