The Pattern Calculus

C. BARRY JAY
University of Technology, Sydney

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 recode for each new type. The pattern calculus and combinatory type system reach this goal by representing each data structure as a combination of 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.

Categories and Subject Descriptors: D.3.3 [**Programming Languages**]: Language Constructs and Features—*Data types and structures; polymorphism*; F.3.3 [**Logics and Meanings of Programs**]: Studies of Program Constructs—*Type structure*

General Terms: Languages, Theory

Additional Key Words and Phrases: Constructor calculus, functional programming, generic programming, pattern calculus, pattern-matching

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* [Backhouse and Sheard 1998; Jeuring 2000; Gibbons and Jeuring 2003] is to support these operations on arbitrary data types without having to recode for each new type. This article 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.

The Université de Paris VII and the University of Edinburgh each hosted me for a month while working on this material, the latter by EP-SRC visiting fellowship No. GR/M36694.
Author's address: University of Technology, Sydney, P.O. Box 123, Broadway NSW 2007, Australia; email: cbj@it.uts.edu.au.

Also, it

—evaluates terms without manipulating types;
—supports a practical type inference algorithm; and
—has been realized in the programming language Bondi.

1.1 Pattern-Matching

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

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

one can define the length of a list by

```
length : List X → Int =
  | Nil → 0
  | Cons h t → length t + 1.
```

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

```
sizetree : Btree_0 X → Int =
  | Leaf_0 → 1
  | Node_0 x y z → 1 + (sizetree y) + (sizetree z).
```

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

```
lengthsize : X → Int =
  | Nil → 0
  | Cons h t → lengthsize t + 1
  | Leaf_0 → 1
  | Node_0 x y z → 1 + (lengthsize y) + (lengthsize z).
```

Although the evaluation of this program is perfectly straightforward, it will not type-check in standard functional languages such as ML [Milner and Tofte 1991; Caml 1997] and Haskell [Haskell 2002] or in the *typed pattern calculus* of Kesner et al. [1996] 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 specialize a default type to one appropriate to its pattern. For example, in lengthsize the default type $X \to \text{Int}$ instantiates X to be either $\text{List}\, Y$ or $\text{Btree}_0\, Y$.

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 *specialization* s. The main type derivation rule for extensions, when stripped of its contexts, is

$$\frac{t : T \to T_2 \quad p : T_1 \quad s : \upsilon T_2}{\text{at } p \text{ use } s \text{ else } t : T \to T_2} \; \upsilon = \mathcal{U}(T_1, T),$$

where υ is the most general unifier of the type T_1 of the pattern and the argument type T of the default function. The specialization need only have type υT_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 specialization 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 specialization 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 specialization attempts to match p_1 with t_1 and p_2 with t_2 (as described in Figure 4).

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

$$\begin{array}{l} \text{bulk} : X \to \text{Int} = \\ \;\mid x\ y \to (\text{bulk}\ x) + (\text{bulk}\ y) \\ \;\mid x \to 1 \end{array}$$

measures the total bulk of its argument, that is, 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

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

Here F is a variable representing a higher type, such as List, which determines a collection of possible structures or *shapes* [Jay 1995, 1996] 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

$$\begin{array}{l}\mathsf{type\ Btree}_2\, X_1 X_2 = \\ \quad \mathsf{Leaf}_2\ \mathsf{of}\ X_1 \\ \mid \mathsf{Node}_2\ \mathsf{of}\ X_2\ \mathsf{and\ Btree}_2\, X_1 X_2\ \mathsf{and\ Btree}_2\, X_1 X_2.\end{array}$$

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

$$\mathsf{map}^2 : (X_1 \to Y_1) \to (X_2 \to Y_2) \to FX_1X_2 \to FY_1Y_2.$$

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

$$\mathsf{type\ Btree}_1\, X = \mathsf{Uniform\ of\ Btree}_2\, XX$$

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

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

$$\begin{array}{l}\mathsf{type\ Btree}(X_1, X_2) = \\ \quad \mathsf{Leaf\ of}\ X_1 \\ \mid \mathsf{Node\ of}\ X_2\ \mathsf{and\ Btree}(X_1, X_2)\ \mathsf{and\ Btree}(X_1, X_2).\end{array}$$

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

$$\begin{array}{l}\mathsf{type}\ A\ \mathsf{has} \\ \quad AXY = \mathsf{Onefun}\ X \to Y \\ \mathsf{or}\ A(X_1, X_2)(Y_1, Y_2) = \mathsf{Bthfun}\ A(X_1, Y_1)\ \mathsf{and}\ A(X_2, Y_2).\end{array}$$

The syntax here introduces a new type A with constructors $\mathsf{Onefun} : (X \to Y) \to AXY$ and $\mathsf{Bthfun} : A(X_1, Y_1) \to A(X_2, Y_2) \to A(X_1, X_2)(Y_1, Y_2)$. 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* [Jay 2001] such as $\forall n.n \to n \to *$.

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

$$\mathsf{map} : \forall F, X, Y.AXY \to FX \to FY.$$

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

$$\mathsf{type\ Nested}\, GFX = \mathsf{Nest\ of}\ G(FX)$$

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., Hindley and Seldin [1986]). 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

$$\mathsf{type}\ KXY = \mathsf{Evr\ of}\ X,$$

where the constructor 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

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

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

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

1.4 Representing Data Types

Two other basic type constants are

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

called the *unit* type and the *parametrized product*, respectively. The constants U, K, I, and B suffice to represent simple polynomial types underpinning List

$$\begin{array}{llll}
\textbf{Type contexts } (\Delta) & \dfrac{}{\vdash} & & \dfrac{\Delta \vdash}{\Delta, X \vdash}\ X \notin \Delta \\[2ex]
\textbf{Types } (S, T) & \dfrac{}{\Delta \vdash X}\ X \in \Delta & \dfrac{}{\Delta \vdash C} & \dfrac{\Delta \vdash S \quad \Delta \vdash T}{\Delta \vdash ST} \\[2ex]
\textbf{Type Schemes } (\tau) & \dfrac{\Delta \vdash T}{\Delta \vdash_s T} & & \dfrac{\Delta, X \vdash_s \tau}{\Delta \vdash_s \forall X.\tau}
\end{array}$$

Fig. 1. Combinatory types.

and Btree_0 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

$$\begin{array}{l}
\mathsf{Nil} = \mathsf{Tag}\ \mathsf{nm(Nil)}\ (\mathsf{Evr}\ \mathsf{Un}), \\
\mathsf{Cons}\ x\ \ y = \mathsf{Tag}\ \mathsf{nm(Cons)}\ (\mathsf{Bind}\ (\mathsf{Ths}\ x)\ \ y).
\end{array} \tag{1}$$

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

$$\mid \mathsf{Tag}\ n\ t \to \mathsf{Tag}\ n\ (\mathsf{map}\ \ f\ \ t).$$

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 Article

Section 1 introduces the article. 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

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 $X_1, \ldots, X_n$ of distinct *type variables* (meta-variables X and Y). The judgment $\Delta \vdash$ 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 judgment $\Delta \vdash T$ 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 judgment $\Delta \vdash_s \tau$ 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 $X_1, \ldots, X_n$ and τ is a type scheme we may write $\forall\Delta.\tau$ in place of $\forall X_1.\forall X_2 \cdots \forall X_n.\tau$.

The choice of type constants determines the character of the type system. We will require a constant $\mathsf{Function}$ where $\mathsf{Function}ST$ or $S \to T$ 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 [Milner 1978]. 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 capitalized word in sans serif, such as List.

The presence of unnatural types such as $\mathsf{List}\,\mathsf{List}$ 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 Jay [2001].

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 $(\sigma_2\sigma_1)X = \sigma_2(\sigma_1 X)$. 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 specialization 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 $\sigma : \Delta_1 \to \Delta_3$ such that $\sigma S = \sigma T$. A *most general unifier* for them which *fixes* Δ_2 is a unifier υ that fixes Δ_2 such that any other unifier σ for them that fixes Δ_2 factors through υ by some substitution ρ so that $\sigma = \rho\upsilon$. When Δ_2 is the empty context then σ is a *unifier* for S and T and υ 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.*

Term contexts (Γ)
$$\frac{\Delta \vdash}{\Delta;\ \vdash} \qquad \frac{\Delta;\Gamma \vdash \quad \Delta \vdash_s \tau}{\Delta;\Gamma, x:\tau \vdash}\ x \notin \Gamma$$

Patterns (p)
$$\frac{}{X; x:X \vdash_\circ x:X} \qquad \frac{}{\Delta;\ \vdash_\circ c:T}\ c:\forall\Delta.T$$

$$\frac{\Delta;\Gamma \vdash_\circ p : T \to S \quad \Delta_1;\Gamma_1 \vdash_\circ p_1 : T_1 \quad \Delta,\Delta_1;\Gamma,\Gamma_1 \vdash}{\Delta,\Delta_1; v\Gamma_1, v\Gamma \vdash_\circ p\ p_1 : vS}\ v = \mathcal{U}(T_1, T)$$

Terms (s,t)
$$\frac{\Delta;\Gamma \vdash}{\Delta;\Gamma \vdash x:\sigma T}\ \begin{array}{l}\Gamma(x) = \forall\Delta_1.T\\ \sigma:\Delta_1 \to \Delta\end{array} \qquad \frac{\Delta;\Gamma \vdash}{\Delta;\Gamma \vdash c:\sigma T}\ \begin{array}{l}c:\forall\Delta_1.T\\ \sigma:\Delta_1 \to \Delta\end{array}$$

$$\frac{\Delta;\Gamma \vdash s : T \to S \quad \Delta;\Gamma \vdash t:T}{\Delta;\Gamma \vdash s\ t : S}$$

$$\frac{\Delta;\Gamma \vdash t : T \to S \quad \Delta_1;\Gamma_1 \vdash_\circ p : T_1 \quad \Delta,\Delta_1; v\Gamma, v\Gamma_1 \vdash s : vS}{\Delta;\Gamma \vdash \mathsf{at}\ p\ \mathsf{use}\ s\ \mathsf{else}\ t : T \to S}\ v = \mathcal{U}(T_1, T)$$

$$\frac{\Delta;\Gamma \vdash t : T \to S \quad \Delta_1;\Gamma_1 \vdash_\circ p : T_1 \quad \Delta,\Delta_1;\Gamma,\Gamma_1 \vdash}{\Delta;\Gamma \vdash \mathsf{at}\ p\ \mathsf{use}\ s\ \mathsf{else}\ t : T \to S}\ \mathcal{U}(T_1, T)\uparrow$$

$$\frac{\Delta,\Delta_1;\Gamma \vdash s:S \quad \Delta;\Gamma, x:\forall\Delta_1.S \vdash t:T}{\Delta;\Gamma \vdash \mathsf{let}\ x = s\ \mathsf{in}\ t : T}$$

$$\frac{\Delta,\Delta_1;\Gamma, x:\forall\Delta_1.T \vdash t:T}{\Delta,\Delta_1;\Gamma \vdash \mathsf{fix}\ (x,t) : T}$$

Fig. 2. The pattern calculus.

PROOF. When Δ_2 is empty the result is the standard one due to Robinson [1965]. 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. □

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 $\mathcal{U}(\Delta_2, S, T) : \Delta_1\Delta_2 \to \Delta_1\Delta_2$. 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.

The *patterns* (meta-variable p) are either term variables, *constructors* (meta-variable c) or *applications* of one pattern to another. The judgment $\Delta;\Gamma \vdash_\circ p : T$ 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

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

Fig. 3. Free variables.

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 specializations.

The *raw terms* (meta-variables s and t) are built from variables, constructors, applications, extensions, let-terms, and recursion. The judgment $\Delta; \Gamma \vdash t : T$ 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 $t : T$ then the *application* $s\ t$ has type S.

The term at p use s else t is an *extension* of the *default function* t at the pattern p by the *specialization* 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 υ then it suffices to type the specialization by υS in the context obtained by applying υ 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 specialization but not in the default. Note that υ does *not* appear in the conclusion, which will constrain type inference. If T_1 and T do not have a unifier then specialization 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 let $x = s$ 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 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, that is, *bound*, are those occurring free in a pattern, let-bound variables or recursion variables. α-conversion of bound

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

Fig. 4. Reduction rules for the pattern calculus.

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 $\mid p \rightarrow t$ corresponds to the program fragment at p use t 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 nonterminating term of polymorphic type, for example, $\mathsf{fix}\ (x.x)$. In Bondi such errors are defined using an exception constructor so that exception handling can be done by pattern-matching.

A complete term of the form $\mid p \rightarrow t$ can be regarded as a λ-abstraction $\lambda p.t$ of t with respect to the pattern p (see, e.g., Forest [2002]). Also, define $g.f$ to be $\lambda x.g\ (f\ x)$ for some fresh variable x. The wildcard symbol $\_$ in $\mid \_ \rightarrow t$ represents a fresh variable. We may also write match t_1 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 specialize and when to default. Some care is required to avoid a default when further reduction could allow specialization to occur. A *constructed term* is a term t whose head is a constructor, that is, 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 specialization always occurs with the argument t_1 being substituted for x in the specialization, just as in β-reduction of λ-abstractions. When the pattern is a constructor c then if the argument is also c then the specialization 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

```
general_equal : $X \to Y \to$ Bool =
  | $x_1$ $x_2$ $\to$ ( | $y_1$ $y_2$ $\to$ general_equal $x_1$ $y_1$ && (general_equal $x_2$ $y_2$) | $y$ $\to$ False)
  | $x$ $\to$  | $y$ $\to$ primequal $x$ $y$

equal : $X \to X \to$ Bool = general_equal
```

Fig. 5. Equality.

an applicative term $t_1\ t_2$ then specialization occurs. The specialization 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* $t \to_1 t'$ is given by the application of a basic reduction to a subterm of t. A *reduction* $t \to t'$ is given by a finite sequence of one-step reductions $t \to_1 t_1 \to \cdots \to t'$.

Unlike most approaches to generic programming, reduction here does not require explicit type information to guide specialization 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 specialization rules do not.

4. BASIC EXAMPLES

4.1 Booleans

Booleans are declared by

$$\text{type Bool} = \text{True} \mid \text{False}.$$

The standard conditional is defined by

$$\text{if } b \text{ then } s \text{ else } t = \text{match } b \text{ with } \mid \text{True} \to s \mid \text{False} \to t$$

or (at True use s else at False use t else error) b. Other variations are possible, for example, 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 not : Bool $\to$ Bool can be defined in the usual way. It is useful to have a primitive equality

$$\text{primequal} : X \to Y \to \text{Bool}$$

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 afterward when defining equal.

```
(general_plus : $X \to Y \to X$) $u$ $v$ =
  match $(u, v)$ with
    | (Int $x$, Int $y$) $\to$ Int $(x +_i y)$
    | (Float $x$, Float $y$) $\to$ Float $(x + .y)$
    | $(x_1\ x_2, y_1\ y_2)$ $\to$ (general_plus $x_1$ $y_1$) (general_plus $x_2$ $y_2$)
    | $(x, y)$ $\to$ match primequal $x$ $y$ with  | True $\to$ $x$

(plus : $X \to X \to X$) = general_plus
```

Fig. 6. Addition.

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) $\to$
general_equal (Cons True) (Cons True) && (general_equal Nil Nil) $\to$
general_equal Cons Cons && (general_equal True True) $\to$ 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 pInt, 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, for example, 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 pInt,
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 : (pInt $\to$ pInt $\to$ pInt) $\to$ (pFloat $\to$ pFloat $\to$ pFloat) $\to X \to Y \to X$

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.

5. PROPERTIES OF REDUCTION

THEOREM 5.1 (CHURCH-ROSSER). *Reduction is Church-Rosser.*

PROOF. Given a fixed set of constructors, the rules for specialization can be expanded to a set of rules which are left-linear, as well as nonoverlapping (see, for example, Klop [1980]). □

A term is *reducible* if there is a reduction rule that can be applied to one of its subterms. Otherwise it is *in normal form* or *normal*.

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. □

LEMMA 5.3. *Typings of terms are stable under term substitution. That is, if there are derivations $\Delta; \Gamma, x : \forall\Delta_1.S \vdash t : T$ and $\Delta, \Delta_1; \Gamma \vdash s : S$ then there is a derivation of $\Delta, \Delta_1; \Gamma \vdash t\{s/x\} : T$.*

PROOF. The proof is by induction on the structure of the derivation of the typing of t. All of the cases are straightforward. □

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 specializations of extensions as described in Figure 4.

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

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

and a derivation $\Delta; \Gamma \vdash t_1 : T$ to produce $\Delta; \Gamma \vdash (\mathsf{at}\ x\ \mathsf{use}\ s\ \mathsf{else}\ t)\ t_1 : S$. Now X is not free in Γ or $T \to S$ 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 specialization at a constructor. A derivation for the extension must take the form

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

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

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

$$\frac{\Delta_1; \Gamma_1 \vdash_\circ p_1 : T_3 \to T_1 \quad \Delta_2; \Gamma_2 \vdash_\circ p_2 : T_4}{\Delta_1, \Delta_2; \upsilon_1(\Gamma_1, \Gamma_2) \vdash_\circ p_1\ p_2 : \upsilon_1 T_1}\ \upsilon_1 = \mathcal{U}(T_4, T_3).$$

Then the extension must have a type derivation

$$\frac{\begin{array}{c}\Delta; \Gamma \vdash t : T \to S \\ \Delta_1, \Delta_2; \upsilon_1(\Gamma_1, \Gamma_2) \vdash_\circ p_1\ p_2 : \upsilon_1 T_1 \\ \Delta, \Delta_1, \Delta_2; \upsilon\upsilon_1(\Gamma, \Gamma_1, \Gamma_2) \vdash s : \upsilon S\end{array}}{\Delta; \Gamma \vdash \mathsf{at}\ p_1\ p_2\ \mathsf{use}\ s\ \mathsf{else}\ t : T \to S}\ \upsilon = \mathcal{U}(\upsilon_1 T_1, T)$$

whose application to $t_1\ t_2$ requires derivations of $\Delta; \Gamma \vdash t_1 : T_5 \to T$ and $\Delta; \Gamma \vdash t_2 : T_5$ for some type T_5. From these we can create a derivation of $\Delta; \Gamma \vdash \lambda x, y.t\ (x\ \ y) : (T_5 \to T) \to T_5 \to S$. Now consider the extension of this function at p_1. If $\upsilon_2 = \mathcal{U}(T_3 \to T_1, T_5 \to T)$ does not exist then we are done, so assume that it does exist so that it suffices to derive $\Delta, \Delta_1; \upsilon_2(\Gamma, \Gamma_1) \vdash \mathsf{at}\ p_2\ \mathsf{use}\ s\ \mathsf{else}\ \lambda y.t\ (p_1\ y) : \upsilon_2(T_5 \to S)$. The derivation for the default is straightforward. For the specialization, assume that the most general unifier $\upsilon_3 = \mathcal{U}(T_4, \upsilon_2 T_5)$ exists. Then it suffices to derive of $\Delta, \Delta_1, \Delta_2; \upsilon_3\upsilon_2(\Gamma, \Gamma_1, \Gamma_2) \vdash s : \upsilon_3\upsilon_2 S$. Now $\upsilon\upsilon_1$ is the most general unifier of the pairs (T_3, T_4) and (T_1, T). Also, $\upsilon_3\upsilon_2 T_3 = \upsilon_3\upsilon_2 T_5 = \upsilon_3\upsilon_2 T_4$ and $\upsilon_3\upsilon_2 T_1 = \upsilon_3\upsilon_2 T$. Hence, $\upsilon_3\upsilon_2$ factors as $\rho\upsilon\upsilon_1$ for some substitution ρ. Applying ρ to the previous derivation for s achieves the result. □

6. TYPE INFERENCE

The type inference algorithm for the pattern calculus is derived from Milner's algorithm $\mathcal{W}$ [Milner 1978]. Type inference for patterns is straightforward. That for terms is more subtle since extensions (like polymorphic recursions; see e.g., Henglein [1993]) do not always have principal type schemes.

For example, the term $\mid \mathsf{Un} \to \mathsf{Un}$ has type schemes $\forall X.X \to X$ and also $\forall X.X \to U$ which do not admit a common generalization. Desugaring this example produces $\mathsf{at}\ \mathsf{Un}\ \mathsf{use}\ \mathsf{Un}\ \mathsf{else}\ \mathsf{error}$ which might suggest that error is the source of difficulties. However, the recursion-free term

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

is typed by both $(\mathsf{Bool} \to FY) \to \mathsf{Bool} \to FY$ and $(X \to F\mathsf{Bool}) \to X \to F\mathsf{Bool}$ but not their common generalization $(X \to FY) \to X \to FY$ 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* $\sigma : \Delta \to \Delta'$ *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 $\Delta';\sigma\Gamma \vdash t : \sigma(T \to S)$ 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 υ_1. Now $\sigma T_1 = T_1$ and so $\upsilon_1\sigma$ is a unifier of T_1 and T and so factors as $\rho\upsilon$ for some substitution ρ. Applying ρ to the derivation of s yields $\Delta, \Delta_1; \upsilon_1\sigma\Gamma, \upsilon_1\Gamma_1 \vdash s : \upsilon_1\sigma S$ from which the desired derivation follows. □

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 $\mathcal{W}(\Delta_a, \Delta_b, \Gamma, t, T) : \Delta_a \to \Delta', \Delta_b$ 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, for example, 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 $\sigma_1 : \Delta_a, X \to \Delta_1\Delta_b$ be $\mathcal{W}((\Delta_a, X), \Delta_b, \Gamma, t_1, X)$ and $\sigma_2 : \Delta_1 \to \Delta_2, \Delta_b$ be $\mathcal{W}(\Delta_1, \Delta_b, \sigma_1\Gamma, t_2, \sigma_1(X \to T))$ where X is a fresh type variable. Then σ is $\sigma_2\sigma_1$.
(4) t is at p use s else t. Let υ_1 be $\mathcal{U}(\Delta_b, Y_1 \to Y_2, T)$ where Y_1 and Y_2 are fresh variables. Let $\sigma_2 : \Delta_a, Y_1, Y_2 \to \Delta', \Delta_b$ be $\mathcal{W}((\Delta_a, Y_1, Y_2), \Delta_b; \upsilon_1\Gamma, t, \upsilon_1 T)$ and define σ to be $\sigma_2\upsilon_1$. 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 υ_3 be $\mathcal{U}(\Delta_b, T_1, \sigma Y_1)$. If $\mathcal{W}(\ , (\Delta', \Delta_b, \Delta_1), \upsilon_3\sigma(\Gamma, \Gamma_1), s, \upsilon_3\sigma Y_2)$ 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 υ_3.
(5) t is let $x = t_1$ in t_2. Let Y_1 be a fresh type variable and let $\sigma_1 : \Delta_a, Y_1 \to \Delta_1, \Delta_2$ 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 $\sigma_2 : \Delta_1 \to \Delta', \Delta_b$ 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 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* $\sigma : \Delta_a \to \Delta', \Delta_b$ *then* $\Delta', \Delta_b;\sigma\Gamma \vdash t : \sigma T$.

PROOF. The proof is by straightforward induction on the structure of t. □

Evidence for the usefulness of this algorithm is that it is able to type all of the examples in the article using only their given type information.

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 Bird and Meertens [1998] (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 article 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 parametrized product of type $BFGX$ is to apply a function of X while ignoring F and G. In other words, X will be the data parameter when declaring $BFGX$. 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, for example, data types, are defined similarly.

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

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

such that each Z_i for $1 \le i \le q$ is a tuple of type variables each of which is distinct from all others in $DZ_1 \cdots Z_q$ and each $T_{i,j}$ is a type whose free type variables are all free in $DZ_1 \cdots Z_q$. The *arity* of the declaration is q. The variables in Z_i for $1 \le i \le q$ 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 a *declared constructor* whose type scheme is

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

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

$$\text{type } D \text{ has } \mathsf{dec}_1(D) \text{ or } \cdots \text{ or } \mathsf{dec}_m(D),$$

where each $\mathsf{dec}_i(D)$ (for $1 \le i \le m$) 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 *parametrized* declared type. A trivial example of a parameter-free declared type is the *unit type*

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

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 $DT_1 \cdots T_q$ 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 $\mathsf{Tag}\ n\ t'$ where t' is its concrete representation, n is a name, and

$$\mathsf{Tag} : (F \to G) \to FX \to GX$$

$$
\begin{array}{l}
\textsf{type } KXY = \textsf{Evr of } X \\
\textsf{type } IX = \textsf{Ths of } X \\
\textsf{type } BFGX = \textsf{Bind of } FX \textsf{ and } GX \\
\textsf{type } LF(X,Y) = \textsf{Asl of } FX \\
\textsf{type } RF(X,Y) = \textsf{Asr of } FY \\
\textsf{type } S_1GFX = \textsf{Rep of } G(X,FX) \\
\textsf{type } S_2GF(X,Y) = \textsf{Rpp of } G(X,(FX,Y)) \\
\textsf{type } MF(W,(X,(Y,Z))) = \textsf{Mid of } F(W,((X,Y),Z))
\end{array}
$$

Fig. 7. The representing data types.

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, for example, 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.

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) Ys of type KXY. For example, the Nil list has concrete form given by

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

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

$$\textsf{Bind (Ths } x)\ y : BI\ \textsf{List}X.$$

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) : $LI(X, Y)$ and that of Node $x\ y\ z$ is

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

The constructors Rep and Rpp for the types *replicate* S_1 and *parametrized replicate* S_2 are used to represent nested data structures. For example, Nest x : Nested GFX has concrete form Rep (Asr x) : $S_1(RG)FX$. Similarly, Rpp will be used to handle multiple data types. For example, given the declaration

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

```
type $A$ has
   $AXY$ = Onefun of $X \to Y$
or $A(X_1, X_2)(Y_1, Y_2)$ = Bthfun of $A(X_1, Y_1)$ and $A(X_2, Y_2)$

(map : $AXY \to FX \to FY$) $f$ =
   | Evr $x \to$ Evr $x$
   | Ths $x \to$ (match $f$ with  | Onefun $f_1 \to$ Ths ($f_1$ $x$))
   | Bind $x$ $y \to$ Bind (map $f$ $x$) (map $f$ $y$)
   | Asl $x \to$ (match $f$ with  | Bthfun $f_1$ $f_2 \to$ Asl (map $f_1$ $x$))
   | Asr $x \to$ (match $f$ with  | Bthfun $f_1$ $f_2 \to$ Asr (map $f_2$ $x$))
   | Rep $x \to$ Rep (map (Bthfun $f$ (Onefun (map $f$))) $x$)
   | Rpp $x \to$ (match $f$ with
               | Bthfun $f_1$ $f_2 \to$ Rpp (map (Bthfun $f_1$ (Bthfun (Onefun (map $f_1$)) $f_2$)) $x$))
   | Mid $x \to$ (match $f$ with
               | Bthfun $f_1$ (Bthfun $f_2$ (Bthfun $f_3$ $f_4$)) $\to$
                  Mid (map (Bthfun $f_1$ (Bthfun (Bthfun $f_2$ $f_3$) $f_4$)) $x$))
   | Tag $n$ $x \to$ Tag $n$ (map $f$ $x$)

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

Fig. 8. Mapping.

and the term map2 from Figure 8 then the concrete representation of $\mathsf{Uniform}_2$ is

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

Longer tuples of types can be handled by inserting more copies of Rpp. Finally, Mid reorders 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 : $X \to Y \to X * Y$ 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

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

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

The pattern for Rep : $G(X, FX) \to S_1GFX$ is typical of the more complex cases. As map f has type $FX \to FY$ then Bthfun f (Onefun (map f)) has type $A(X, FX)(Y, FY)$), 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* $e_t[G, Z, T] : G(Z, T) \to F_t[G, Z, T]Z$ 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

$$\mathsf{Rep} : G(Z, FZ) \to S_1 GFZ.$$

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

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

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

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

Given Z as above, each data type T has an *extractor* $e[T, Z] : T \to F[T, Z]Z$ 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 $\mathsf{Evr} : T \to KTZ$. 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

$$\mathsf{Ths} : Z \to IZ.$$

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

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

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

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

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 $\mathsf{Onefun}\ e[T', Z]$. If it is a pair (T'_1, T'_2) then define f to be $\mathsf{Bthfun}\ f_1\ f_2$ 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

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

8.4 Defining Abstractors

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

$$\mathsf{concr}_n(a) = e[T_n, Z] : T_n \to F[T_n, Z]Z,$$

$$\mathsf{concr}_i(a)\ x_i\ \cdots\ x_n = \mathsf{Bind}(e[T_i, Z]\ x_i)(\mathsf{concr}_{i+1}(a)\ x_{i+1}\ \cdots\ x_n) : BF[T_i, Z]F_{i+1}(a)Z.$$

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

$$a = \lambda x_1, \ldots, x_n.\mathsf{Tag}\ \mathsf{nm}(a)\ (\mathsf{concr}(a)\ x_1\ \cdots\ x_n).$$

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

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

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 generalize 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 $y_1, \ldots, y_n$ produces $f\ (\cdots(f\ x\ y_1)\cdots)\ y_n$. We can use it to define the size of data structure by

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

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 $f_i : X \to Y_i \to X$ for $i = 1, 2$ into a single structure using ParamOne and ParamBth. Similarly, one can define ArrowParam dually to ParamArrow and foldright : ArrowParam $XY \to FX \to Y \to Y$ whose only essential difference from foldleft is that folding over some Bind $x_1\ x_2$ acts on x_2 first and then x_1.

```
type ParamArrow has
  ParamArrow $XY$ = ParamOne of $X \to Y \to X$
or ParamArrow $X(Y_1, Y_2)$ = ParamBth of ParamArrow $XY_1$ and ParamArrow $XY_2$

(foldleft : ParamArrow $XY \to X \to FY \to X$) $f$ $x$ $u$ =
match $(u, f)$ with
   | (Evr $y$, _) $\to x$
   | (Ths $y$, ParamOne $f_1$) $\to f_1\ x\ y$
   | (Bind $y_1\ y_2$, _) $\to$ foldleft $f$ (foldleft $f\ x\ y_1$) $y_2$
   | (Asl $y$, ParamBth $f_1\ f_2$) $\to$ foldleft $f_1\ x\ y$
   | (Asr $y$, ParamBth $f_1\ f_2 \to$ foldleft $f_2\ x\ y$
   | (Rep $y$, _) $\to$ foldleft (ParamBth $f$ (ParamOne (foldleft $f$))) $x\ y$) $f$
   | (Rpp $y$, ParamBth $f_1\ f_2$) $\to$
        foldleft (ParamBth $f_1$ (ParamBth (ParamOne (foldleft $f_1$)) $f_2$)) $x\ y$
   |  (Mid $y$, ParamBth $f_1$ (ParamBth $f_2$ (ParamBth $f_3\ f_4$))) $\to$
        foldleft (ParamBth $f_1$ (ParamBth (ParamBth $f_2\ f_3$) $f_4$)) $x\ y$
   | (Tag $n\ y$, _) $\to$ foldleft $f\ x\ y$

(foldleft1 : $(X \to Y \to X) \to X \to FY \to X$) $f$ = foldleft (ParamOne $f$)
```

Fig. 9. foldleft.

```
type Arrow2 has
  Arrow2 $XYZ$ = Onefun2 of $X \to Y \to Z$
or Arrow2$(X_1, X_2)(Y_1, Y_2)(Z_1, Z_2)$ = Bthfun2 of Arrow2 $X_1Y_1Z_1$ and Arrow2 $X_2Y_2Z_2$

(general_zipwith : Arrow2$XYZ \to FX \to GY \to FZ$) $f\ u\ v$ =
match $(u, v, f)$ with
   | (Evr $x$, Evr $y$, _) $\to$ ( |  True $\to$ Evr $x$) (general_equal $x\ y$)
   | (Ths $x$, Ths $y$, Onefun2 $f_1$) $\to$ Ths ($f_1\ x\ y$)
   | (Bind $x_1\ x_2$, Bind $y_1\ y_2$, _) $\to$ Bind (general_zipwith $f\ x_1\ y_1$) (general_zipwith $f\ x_2\ y_2$)
   | (Asl $x$, Asl $y$, Bthfun2 $f_1\ f_2$) $\to$ Asl (general_zipwith $f_1\ x\ y$)
   | (Asr $x$, Asr $y$, Bthfun2 $f_1\ f_2$) $\to$ Asr (general_zipwith $f_2\ x\ y$)
   | (Rep $x$, Rep $y$, _) $\to$ Rep (general_zipwith (Bthfun2 (Onefun2 (general_zipwith $f$))) $x\ y$)
   | (Rpp $x$, Rpp $y$, Bthfun2 $f_1\ f_2$) $\to$
        Rpp (general_zipwith (Bthfun2 $f_1$ (Bthfun2 (Onefun2 (general_zipwith $f_1$) $f_2$))) $x\ y$)
   | (Mid $x$, Mid $y$, Bthfun2 $f_1$ (Bthfun2 $f_2$ (Bthfun2 $f_3\ f_4$))) $\to$
        Mid (general_zipwith (Bthfun2 $f_1$ (Bthfun2 (Bthfun2 $f_2\ f_3$) $f_4$)) $x\ y$)
   | (Tag $m\ x$, Tag $n\ y$, _) $\to$ match primequal $m\ n$ with  | True $\to$ Tag $m$ (general_zipwith $f\ x\ y$)

(zipwith : Arrow2 $XYZ \to FX \to FY \to FZ$) = general_zipwith
(zipwith1 : $(X \to Y \to Z) \to FX \to FY \to FZ$) $f$ = zipwith (Onefun2 $f$)
```

Fig. 10. zipwith.

Now consider zipping as defined in Figure 10. The type Arrow2 is used to represent tuples of functions. On lists, zipwith1 takes a function $f : X \to Y \to Z$ and a pair of lists, one of Xs and one of Ys and produces a list whose entries are given by applying f to corresponding list entries. Both lists must have the same 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** [Girard et al. 1989] whose types

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

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

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

That is, given a value of type $X + Y$ and two functions of types $X \to Z$ and $Y \to Z$, 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

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

That is, the coproduct is defined by its *behavior* 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 [Kesner et al. 1996] 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 $\lambda(\mathsf{Nil} \mid_\xi \mathsf{Cons}\, x\, xs).[M \mid_\xi N]$ where ξ is used to relate Nil to M and Cons x xs to N. This mechanism is further developed in Forest and Kesner [2003].

11.3 Generic Programming

Most other treatments of generic programming either focus on the semantics [Meijer et al. 1991; Cockett and Fukushima 1992], or use type information to drive the evaluation [Jay and Cockett 1994; Abadi et al. 1995; Jansson and Jeuring 1997; Jansson 2000; Hinze 2000, 2002]. 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 [Hinze 2002] 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, preprocessing of the whole program is used to specialize 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 [Jay et al. 1998] 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

$$\mathsf{intrl} : F(X, (\mu F)X) \to (\mu F)X,$$

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 $\mathsf{type}\ D\ X = \cdots$ 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* [Jay 2001], which uses a weaker form of extension with the syntax

$$\mathsf{under}\ c\ \mathsf{apply}\ f\ \mathsf{else}\ g,$$

where c is a constructor. If c takes n arguments then this can be translated to $\mathsf{at}\ c\ x_1 \cdots x_n\ \mathsf{use}\ f\ x_1 \cdots x_n\ \mathsf{else}\ g$. 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 $ct_1 \cdots t_n$ 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 article has been to demonstrate the expressive power of extensions in a purely functional setting. Successors to this article will show how to add extra features. For example, adding some primitive imperative features is enough to support the definition of a generic assignment

$$\mathsf{assign} : \forall X.\mathsf{loc}\ \ X \to X \to \mathsf{Comm},$$

where $\mathsf{loc}\ \ X$ 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., Hammond and Michaelson [1999]).

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., Barr and Wells [1990]) but the standard categorical semantics typically represents 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 Bondi. The implementation is robust enough to type all of the examples in this article 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.

ACKNOWLEDGMENTS

I would like to thank my hosts Pierre-Louis Curien at Université de Paris VII and Don Sannella at the University of Edinburgh 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, Bruno Marnette, 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

Abadi, M., Cardelli, L., Pierce, B., and Rémy, D. 1995. Dynamic typing in polymorphic languages. *J. Funct. Programm. 5*, 1, 111–130.

Backhouse, R. and Sheard, T., Eds. 1998. *Workshop on Generic Programming: Marstrand, Sweden, 18th June, 1998*. Chalmers University of Technology.

Barr, M. and Wells, C. 1990. *Category Theory for Computing Science*. International Series in Computer Science. Prentice Hall, Englewood Cliffs, NJ.

Bird, R. and Meertens, L. 1998. Nested datatypes. In *Proceedings of the 4th International Conference on Mathematics of Program Construction (MPC'98, Marstrand, Sweden, 15–17 June)* J. Jeuring, Ed. Lecture Notes in Computer Science, vol. 1422. Springer-Verlag, Berlin, Germany, 52–67.

Caml. 1997. Objective Caml. Available online at `http://pauillac.inria.fr/ocaml`.

Cockett, J. and Fukushima, T. 1992. About charity. Tech. rep. 92/480/18, University of Calgary, Calgary, Alta., Canada.

Forest, J. 2002. A weak calculus with explicit operators for pattern matching and substitution. In *Rewriting Techniques and Applications, 13th International Conference, RTA 2002, Copenhagen, Denmark, July 22–24, 2002, Proceedings*, S. Tison, Ed. Vol. 2378. Springer, Berlin, Germany, 174–191.

Forest, J. and Kesner, D. 2003. Expression reduction systems with patterns. In *14th International Conference on Rewriting Techniques and Applications*, R. Nieuwenhuis, Ed. Lecture Notes in Computer Science, vol. 2706. Springer Verlag Berlin, Germany.

Gibbons, J. and Jeuring, J., Eds. 2003. *Generic Programming: IFIP TC2/WG2.1 Working Conference on Generic Programming July 11–12, 2002, Dagstuhl, Germany*. Kluwer Academic Publishers, Dordrecht, The Netherlands.

Girard, J.-Y., Lafont, Y., and Taylor, P. 1989. *Proofs and Types*. Tracts in Theoretical Computer Science. Cambridge University Press, Cambridge, U.K.

Hammond, K. and Michaelson, G. E. 1999. *Research Directions in Parallel Functional Programming*. Springer, Berlin, Germany.

Haskell. 2002. Haskell. Available online at http://www.haskell.org/.

Henglein, F. 1993. Type inference with polymorphic recursion. *ACM Trans. Programm. Lang. Sys. 15*, 253–289.

Hindley, R. and Seldin, J. 1986. *Introduction to Combinators and Lambda-Calculus*. Cambridge University Press, Cambridge, U.K.

Hinze, R. 2000. A new approach to generic functional programming. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, New York, NY, 119–132.

HINZE, R. 2002. Polytypic values possess polykinded types. *Sci. Comput. Programm. 43*, 129–159.

JANSSON, P. 2000. Functional polytypic programming. Ph.D. dissertation. Chalmers University, Göteborg, Sweden.

JANSSON, P. AND JEURING, J. 1997. PolyP—a polytypic programming language extension. In *Proceedings of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, New York, NY, 470–482.

JAY, C. 1995. A semantics for shape. *Sci. Comput. Programm. 25*, 251–283.

JAY, C. 1996. Shape in computing. *ACM Comput. Surv. 28*, 2, 355–357.

JAY, C. 2001. Distinguishing data structures and functions: The constructor calculus and functorial types. In *Typed Lambda Calculi and Applications: 5th International Conference TLCA 2001, Kraków, Poland, May 2001 Proceedings*, S. Abramsky, Ed. Lecture Notes in Computer Science, vol. 2044. Springer, Berlin, Germany, 217–239.

JAY, C., BELLÈ, G., AND MOGGI, E. 1998. Functorial ML. *J. Funct. Programm. 8*, 6, 573–619.

JAY, C. AND COCKETT, J. 1994. Shapely types and shape polymorphism. In *Programming Languages and Systems—ESOP '94: 5th European Symposium on Programming, Edinburgh, U.K., April 1994, Proceedings*, D. Sannella, Ed. Lecture Notes in Computer Science, vol. 788. Springer-Verlag, Berlin, Germany, 302–316.

JEURING, J., Ed. 2000. *Proceedings: Workshop on Generic Programming (WGP 2000): July 6, 2000, Ponte de Lima, Portugal*. Publication, UU-CS-2000-19, Utrecht University, Utrecht, The Netherlands.

KESNER, D., PUEL, L., AND TANNEN, V. 1996. A Typed Pattern Calculus. *Inform. Computat. 124*, 1, 32–61.

KLOP, J. 1980. Combinatory reduction systems. Ph.D. dissertation. Mathematical Center Amsterdam, Amsterdam, The Netherlands. Tracts 129.

MEIJER, E., FOKKINGA, M., AND PATERSON, R. 1991. Functional programming with bananas, lenses, envelopes and barbed wire. In *Procceding of the 5th ACM Conference on Functional Programming and Computer Architecture*, J. Hughes, Ed. Lecture Notes in Computer Science, vol. 523. Springer Verlag, Berlin, Germany, 124–44.

MILNER, R. 1978. A theory of type polymorphism in programming. *J. Comput. Syst. Sci. 17*, 3, 348–375.

MILNER, R. AND TOFTE, M. 1991. *Commentary on Standard ML*. MIT Press, Cambridge, MA.

ROBINSON, J. 1965. A machine-oriented logic based on the resolution principle. *J. Assoc. Comput. Mach. 12*, 23–41.

Received November 2001; revised July 2002, June 2003; accepted November 2003