Type Parametric Programming

Tim Sheard
Pacific Software Research Center
Oregon Graduate Institute of Science & Technology
20000 NW Walker Road
Portland, OR 97291-1000
sheard@cse.ogi.edu

November 30, 1993

Abstract

We introduce a new abstraction mechanism, *type parametric combinators*, which supports abstraction over type constructors defined by datatype declarations found in functional languages such as Miranda, Haskell, and ML. This mechanism allows the definition and use of high level abstractions not possible in traditional languages and could be used to define *user programmable* derived instance declarations for type classes in Haskel. We illustrate its use in an actual programming language by giving examples in the ML dialect CRML.

1 Introduction

Abstraction is the key to effective programming. Abstracting over values leads to functions, abstracting over type parameters leads to parametric polymorphism. The thesis of this paper is that abstracting over type constructors leads to even more effective programming.

The *datatype* declaration in ML introduces a new *type* constructor and a set of *value* constructors which have instances of this type constructor as their range. Type parametric programming allows programmers to construct algorithms which operate over values of large classes of datatypes where the class has some particular properties. For example, a structural equality function can be defined for datatypes whose value constructors have no functions in their domain.

Such ideas were originated by category theorists. They define types as the least fixed point of a functor, then use this functor to capture recursion patterns over that type[4, 3, 9]. In our work we define types using recursive type equations, in the manner of functional programmers, and then derive the functor using type parametric combinators. This has the distinct advantage of being familiar to a larger audience, and allows a richer class of recursion patterns to be defined, notably binary functions.

Type parametric combinators allow higher level abstractions than found in traditional languages, and unify several ad hoc methods found in existing languages such as *eq-types* in ML, and *derivable type classes* in Haskell. These ad hoc methods are limited becuase they cannot be extended by the programmer.

In Sections 2 and 3 we define a class of types and define the notion of type parametric combinators for this class. In Section 4 we use these combinators to construct *templates* for a number of interesting classes of functions. In Section 5 we illustrate how theorems can be proved for all functions generated by a particular template. In Section 6 we illustrate how type parametric combinators could be used to address the longstanding problem of *equality types* in ML and extend *derived type classes* in Haskell. In Section 7 we extend our techniques to mutually recursive types. In Section 8 we illustrate how these techniques can be easily added to traditional functional languages by the use of a phase distinction. We illustrate this by describing an implementation we have built on top of ML we call CRML. Finally, in Section 9 we conclude.

2 Types

Definition 1 (Type Constructors and Datatypes) *The datatypes we consider are tagged sums defined by recursive type equations of the form:*

$$T(\alpha_1, \ldots, \alpha_p) = C_1(t_1) \;|\; \cdots \;|\, C_n(t_n)$$

where T is the type constructor, $\alpha_1, \ldots, \alpha_p$ denote type variables, the $C_i : t_i \rightarrow T(\alpha_1, \ldots, \alpha_p)$ are names of value constructors (the tags), and the t_i are either:

- *primitive types such as int or string*
- *type variables (in the set $\alpha_1, \ldots, \alpha_p$)*
- *type formulas constructed using cross, $(t_1 \times t_2)$, and arrow, $(t_1 \rightarrow t_2)$*
- *instantiations of defined types other than T*
- *the instantiation, $T(\alpha_1, \ldots, \alpha_p)$, of T itself.*

and the type $T(\alpha_1, \ldots, \alpha_p)$ appears in only positive positions in each t_i.

These types correspond to a large subset of the valid *datatypes* definable in ML. For example, the following are tagged sum type definitions:

$$\begin{array}{lcl}
\mathrm{list}(\alpha) & = & \mathrm{Nil} \;|\; \mathrm{Cons}(\alpha \times \mathrm{list}(\alpha)) \\
\mathrm{tree}(\alpha,\beta) & = & \mathrm{Tip}(\alpha) \;|\; \mathrm{Node}(\beta \times \mathrm{tree}(\alpha,\beta) \times \mathrm{tree}(\alpha,\beta)) \\
\mathrm{bush}(\alpha) & = & \mathrm{Leaf}(\alpha) \;|\; \mathrm{Fork}(\mathrm{list}(\mathrm{bush}(\alpha))) \\
\mathrm{nat} & = & \mathrm{Zero} \;|\; \mathrm{Succ}(\mathrm{nat}) \\
\mathrm{term}(\alpha) & = & \mathrm{Var}(\alpha) \;|\; \mathrm{Lam}(\alpha \times \mathrm{term}(\alpha)) \;|\; \mathrm{Apply}(\mathrm{term}(\alpha) \times \mathrm{term}(\alpha))
\end{array}$$

We assume that nullary constructors, like Nil, have domain, $\mathbf{1}$, the type of the unique value of the empty product ().

3 Combinators, Functors

A combinator is an operator that combines or transforms functions to create new functions. Composition, $f \circ g$, is an example of an infix combinator.

A *combinator*, $C(f_1, \ldots, f_n)$, is a *functor* if it preserves identities and composition. That is:

$$\begin{array}{rcll} C(id_1, \ldots, id_n) & = & id & \wedge \\ C(f_1, \ldots, f_n) \circ C(g_1, \ldots, g_n) & = & C(f_1 \circ g_1, \ldots, f_n \circ g_n) & \end{array}$$

A *type parametric combinator* is a combinator that takes a *type* as a parameter in addition to its functional parameters and combines its functional parameters in ways that depend upon its *type* parameter. Their intended use is as compile-time operators that construct functions applicable to objects with a fixed static type. For example consider the simple combinator over the functional parameters f and g:

$$\begin{array}{lcl} C[f, g, t_1 \times \ldots \times t_n] & = & f \\ C[f, g, u \rightarrow v] & = & g \\ C[f, g, \_] & = & id \end{array}$$

It returns f if its type parameter is a product, g if it is an arrow, and the identity function otherwise. A *type parametric functor* is a type parametric combinator that preserves identities and composition for all types t:

$$\begin{array}{rcll} C(id, t) & = & id & \wedge \\ C(f, t) \circ C(g, t) & = & C(f \circ g, t) & \end{array}$$

3.1 The Functor E

We are interested in using type parametric combinators to describe patterns of recursion for an arbitrary type. Functions over T are typically defined by using n equations, one for each constructor of T. Consider the type: $\mathrm{list}(\alpha) = \mathrm{Nil} \mid \mathrm{Cons}(\alpha \times \mathrm{list}(\alpha))$. A function g over lists often has 2 equations, one of which has the form:

$$g\ (Cons(x, xs)) \ = \ \ldots\ (g\ xs)\ \ldots$$

For any constructor function C_i the functor $E_i(f, g)$ (which we will define shortly) captures precisely this notion of "pushing" the function g onto C_i's recursive parameters. An intuitive understanding of E can be obtained by inspecting the formulas:

$list(\alpha) =$ $\quad Nil \mid$ $\quad Cons(\alpha \times list(\alpha))$	$E_{Nil}(f, g) = id_{\mathbf{1}}$ $E_{Cons}(f, g) = f \times g$

For each constructor C_i, the right-hand side of the definition of E_i is formed from C_i's domain as follows: each constant type t is replaced by id_t, each occurence of $list(\alpha)$ is replaced by g, each occurence of α is replaced by f. The operator $\times$ is lifted to operate on functions as well as types by the rule $(h_1 \times \ldots \times h_n)(x_1, \ldots x_n) \ = \ (h_1\ x_1, \ldots, h_n\ x_n)$. Note the two styles of subscripting. For a fixed constructor we often subscript E with the constructor name rather than with its index. For example, writing E_{Nil} rather than E_1, using E_i when we talk about an arbitrary constructor.

This construction is captured precisely for all tagged sum types by the type parametric combinator M.

Definition 2 (The Type Parametric Combinator M) *Let $T(\overline{\alpha})$ be the left hand side of some type equation then $M^{T(\overline{\alpha})}$ is defined by:*

$$
\begin{aligned}
M^{T(\overline{\alpha})}[\overline{f}, g, \alpha_k] &= f_k && (1)\\
M^{T(\overline{\alpha})}[\overline{f}, g, T(\overline{\alpha})] &= g && (2)\\
M^{T(\overline{\alpha})}[\overline{f}, g, S(t_1, \ldots, t_q)] &= \mathrm{map}^S\ (M^{T(\overline{\alpha})}[\overline{f}, g, t_1], \ldots, M^{T(\overline{\alpha})}[\overline{f}, g, t_q]) && (3)\\
M^{T(\overline{\alpha})}[\overline{f}, g, t_1 \times \ldots \times t_n] &= (M^{T(\overline{\alpha})}[\overline{f}, g, t_1]) \times \ldots \times (M^{T(\overline{\alpha})}[\overline{f}, g, t_n]) && (4)\\
M^{T(\overline{\alpha})}[\overline{f}, g, u \to v] &= \lambda\, h.\ (M^{T(\overline{\alpha})}[\overline{f}, g, v]) \circ h \circ (M^{T(\overline{\alpha})}[\overline{f}, g, u]) && (5)\\
M^{T(\overline{\alpha})}[\overline{f}, g, \_] &= id && (6)
\end{aligned}
$$

where $\overline{\alpha}$ is $\alpha_1, \ldots, \alpha_p$, and $\overline{f}$ is $f_1, \ldots, f_p$, and $S(t_1, \ldots, t_q)$ is the instantiation of some (previously) defined type other than T.

Note that Equation 3 depends upon map^S, which we have not yet defined. The map for S can be defined for any tagged sum type. We will do this shortly in Definition 4.

For a given constructor function, $C_i : t_i \to T(\overline{\alpha})$, both $T(\overline{\alpha})$ and the domain, t_i, are fixed. By abstracting over this information we obtain E that will "push" each f_i into the components with type α_i, and g into the component with type $T(\overline{\alpha})$ [5, 1, 13].

Definition 3 (The Functor E) *For each constructor, $C_i : t_i \to T(\overline{\alpha})$, of T we define:*

$$E_i^T(\overline{f}, g) \;=\; M^{T(\overline{\alpha})}[\,\overline{f},\, g,\, t_i\,]$$

When it is clear from the context, or we are talking about an arbitrary type T, we often omit the superscript, writing E_i rather than E_i^T.

Theorem 1 (Both M and E are functors) *If for all constructors, $C_i : t_i \to T$, T appears only covariantly in t_i [10, 2], then the type parametric combinator, M^T, is a functor, and E^T is a functor as well.*

Proof: *That M is a functor can be shown by induction over the structure of the types* t *(omitted). Then since each E_i is defined to be M with some fixed parameter, E is a functor too.*

A function over a value of type T, where T has N value constructors, can be defined by N recursive equations. Type parametric combinators, like M, are used to "push" the recursive calls onto the correct arguments. A *Template* is a pattern which describes each equation in an manner which abstracts over the type constructors.

4 Templates

We use combinators and functors in templates to define a class of functions for large classes of tagged sum types. The functor E^S is used to generate the *map* for S (that was used in equation 3).

Definition 4 (The Map Template) *The template for* map^S *is:*

$$(\mathrm{map}^S(f_{\alpha_1}, \ldots, f_{\alpha_p})) \circ C_i \;=\; C_i \circ (E_i^S(f_{\alpha_1}, \ldots, f_{\alpha_p}, \mathrm{map}^S(f_{\alpha_1}, \ldots, f_{\alpha_p})))$$

We call the equation above a *template*, since it describes how to construct the equations that make up the functions body.

$list(\alpha) =$ $Nil \;\;|$ $Cons(\alpha \times list(\alpha))$	 $E_{Nil}(f,g) = id_{\mathbf{1}}$ $E_{Cons}(f,g) = f \times g$	 $\mathrm{map}^S\ h\ (Nil()) = Nil()$ $\mathrm{map}^S\ h\ (Cons(x,y)) = Cons(h\ x, \mathrm{map}^S\ h\ y)$
$bush(\alpha) =$ $Leaf(\alpha) \;\;|$ $Fork(list(bush(\alpha)))$	 $E_{Leaf}(f,g) = f$ $E_{Fork}(f,g) = map^{list}\ g$	 $\mathrm{map}^S\ h\ (Leaf\ x) = Leaf(h\ x)$ $\mathrm{map}^S\ h\ (Fork\ L) = Fork(map^{list}(\mathrm{map}^S\ h)\ L)$

Note that the *map* template and M are mutually recursive. Since no type may be defined in terms of a type not yet defined, this recursion is well founded. See Section 7 for methods of handling mutually recursive types.

Ocasionally a single pattern in a template will not suffice to define a class of functions. We illustrate this for the class we call the zero replacement functions. If a datatype definition has a unique nullary constructor, Z, (Nil or $Zero$ are examples of zeroes for types described above) then a zero replacement, $ZR\ y\ x$, replaces all zeros in x with y.

Definition 5 (The Zero Replacement template) *The template for* ZR^S *is:*

$$\begin{aligned}(\mathrm{ZR}^S\ y) \circ Z &\;=\; (Ky) \\ (\mathrm{ZR}^S\ y) \circ C_i &\;=\; C_i \circ (E_i^S(id, \mathrm{ZR}^S\ y)\end{aligned}$$

where Z *is the unique nullary constructor and* K *is the constant combinator such that:* $K\ y\ z = y$.

This example extends the template notation to one which is composed of a sequential list of template patterns. For each constructor the template mechanism attempts to match the patterns in the constructor position of the applicable template equation to the actual constructors. If this is accomplished then the template equation is applied. If this fails the template mechanism moves onto, and tries, the next template equation. It is an error for all template equations to fail.

Note that the function append is a zero replacement for *list*, and that natural number addition is a zero replacement for *nat*.

The well known fold or reduce function for lists can be generalized for any type by the use of templates.

Definition 6 (The Fold Template) *The generic fold (catamorphism [9]) for T is defined by the following set of equations, one for each constructor C_i of T:*

$$\mathrm{fold}^T(\overline{h}) \circ C_i = h_i \circ (E_i^T(\overline{id}, \mathrm{fold}^T(\overline{h})))$$

where $\overline{h} = h_1, \ldots, h_n$ and $\overline{id}$ is the vector of identity functions $id_{\alpha_1}, \ldots, id_{\alpha_p}$.

$list(\alpha) =$ $\quad Nil \;\mid$ $\quad Cons(\alpha \times list(\alpha))$	$E_{Nil}(f, g) = id_{\mathbf{1}}$ $E_{Cons}(f, g) = f \times g$	fold $\overline{h}$ $Nil = h_{Nil}()$ fold $\overline{h}$ $(Cons(x, y)) = h_{Cons}(x, \mathrm{fold}\ \overline{h}\ y)$ where $\overline{h} = (h_{Nil}, h_{Cons})$
$bush(\alpha) =$ $\quad Leaf(\alpha) \;\mid$ $\quad Fork(list(bush(\alpha)))$	$E_{Leaf}(f, g) = f$ $E_{Fork}(f, g) = map^{list}\ g$	fold $\overline{h}$ $(Leaf\ x) = h_L(x)$ fold $\overline{h}$ $(Fork(L)) = h_F(map^{list}(\mathrm{fold}\ \overline{h})\ L)$ where $\overline{h} = (h_L, h_F)$

5 Theorems About Type Parametric Programs

When a type paramtetric combinator has particular properties, complex theorems can often be proven about *all* functions defined using that combinator. For example, because H and E are functors, we may prove a number of interesting things about functions defined in templates using E.

Theorem 2 (map^S is a Functor) *For each type constructor, S, the function generated by the map template, map^S, is a functor.*

Proof: *We need to prove $map\ id = id$ and $(map\ f) \circ (map\ g) = map(f \circ g)$. Let C_i be an arbitrary constructor of S, assume as induction hypothesis $E_i(f, map\ id) = E_i(f, id)$ then:*

$$\begin{array}{lr}
(map\ id) \circ C_i & \\
= C_i \circ E_i(id, map\ id) & \textit{definition} \\
= C_i \circ E_i(id, id) & \textit{ind. hyp.} \\
= C_i \circ id & E_i \textit{ is a functor} \\
= C_i & \textit{property of id}
\end{array}$$

Assume as induction hypothesis $E_i(h, map\ (f \circ g)) = E_i(h, (map\ f) \circ (map\ g))$ then:

$$\begin{array}{lr}
(map\ (f \circ g)) \circ C_i & \\
= C_i \circ E_i(f \circ g, map(f \circ g)) & \textit{definition} \\
= C_i \circ E_i(f \circ g, (map\ f) \circ (map\ g)) & \textit{ind. hyp.} \\
= C_i \circ E_i(f, map\ f) \circ E_i(g, map\ g) & E_i \textit{ is a functor} \\
= (map\ f) \circ C_i \circ E_i(g, map\ g) & \textit{definition} \\
= (map\ f) \circ (map\ g) \circ C_i & \textit{definition}
\end{array}$$

For a template with multiple patterns, like ZR, we will need two cases one for each of the patterns in the template. Consider all functions h such that $h(x, y) = (ZR\ y\ x)$. It is easy to prove that all such functions are assoicative, i.e. $h(h(x, y), z) = h(x, h(y, z))$. This can be cast in terms of ZR as $(ZR\ z)(ZR\ y\ x)\ =\ (ZR\ (ZR\ z\ y)\ x)$.

Theorem 3 **(ZR^S is associative)** *Let C_i be an arbitrary constructor of S. Then we need to prove for all w: $(ZR\ z)(ZR\ y\ (C_i\ w)) = (ZR\ (ZR\ z\ y)\ (C_i\ w))$ or $(ZR\ z) \circ (ZR\ y) \circ C_i\ = (ZR\ (ZR\ z\ y)) \circ C_i$*

Proof: *Case 1: $C_i = z$ where z is the unique nullary constructor.*

$$\begin{array}{lll} = & (ZR\ z) \circ (ZR\ y) \circ z & \\ = & (ZR\ z) \circ (K\ y) & \textit{definition} \\ = & K(ZR\ z\ y) & \textit{pointwise reasoning about } K\ \&\ ZR \\ = & (ZR\ (ZR\ z\ y)) \circ z & \end{array}$$

Case 2: C_i is any other constructor. Assume as induction Hypothesis $E_i(f, (ZR\ z) \circ (ZR\ y)) = E_i(f, (ZR\ (ZR\ z\ y)))$

$$\begin{array}{lll} = & (ZR\ z) \circ (ZRy) \circ C_i & \\ = & (ZR\ z) \circ C_i \circ E_i(id, ZRy) & \textit{definition } ZR \\ = & C_i \circ E_i(id, ZR\ z) \circ E(id, ZRy) & \textit{defintion } ZR \\ = & C_i \circ E_i(id \circ id, (ZR\ z) \circ (ZRy)) & E \textit{ is a functor} \\ = & C_i \circ E_i(id, (ZR\ (ZR\ z\ y))) & \textit{induction Hypothesis} \\ = & (ZR\ (ZR\ z\ y)) \circ C_i & \textit{definition } ZR \end{array}$$

6 Extending Deriveable Type Classes

In ML, equality functions are generated automatically for the the *eq-types*. In Haskel[8] equality is one of the derivable type classes[12] that can be automatcially generated. In both cases such capability is built into the compiler and is not extendable by the programmer. Type parametric combinators can provide a user extendible mechanism. To illustrate this we give a type parametric combinator and template the derives equality functions for all datatypes not containg functions.

6.1 Equality

Consider an equation for a function that recurses over a pair of constructed arguments simultaneously, such as an equality function. One of its parameters will be a pair of constructors, $(C_i\,\bar{x},\ C_j\,\bar{y})$. A a binary template for such a function will contain a product of constructors, $(C_i \times C_j)$. For a type with N constructors, the template will describe $N * N$ equations, one for each distinct pair of constructors.

An equality function for a type $T(\overline{\alpha})$ will be parametrically polymorphic, and will take a vector of equality functions, $\bar{f}$, as an argument as well as a pair of objects being compared.

These parameter functions test equality over the types $\overline{\alpha}$. For example consider the *list* equality function:

$$\begin{array}{lcl} equal^{list}\ (f)\ (Nil, Nil) & = & true \\ equal^{list}\ (f)\ (Cons(x, xs), Cons(y, ys)) & = & f(x, y) \wedge\ equal^{list}\ (f)\ (xs, ys) \\ equal^{list}\ (f)\ (Cons(x, xs), Nil) & = & false \\ equal^{list}\ (f)\ (Nil, Cons(y, ys)) & = & false \end{array}$$

A template for this class of functions will be built upon a type parametric combinator, H, defined below. Let $C_i : d \to T(\overline{\alpha})$ be a constructor of T. Define $F_i(f, g) = (H^{T(\overline{\alpha})}[\bar{f}, g, d])$. The following template will generate a set of $N * N$ equations that defines the equality function for T:

$$\begin{array}{lcl} (equal\ \bar{f}) \circ (C_i \times C_i) & = & F_i(\bar{f}, (equal\ \bar{f})) \\ (equal\ \bar{f}) \circ (C_i \times C_j) & = & K\ false \end{array}$$

In the template above the first template equation matches only if the constructors are identical. In this case the combinator F_i is used. Otherwise, the second equation, which returns the constant function that always returns $false$, is used.

The type parametric combinator, H, is given below. Note that is not a total function, it does not handle arrow types. Thus the template for equalities is applicable to a large class of constructed types, but not all types.

Let eq_{int}, eq_{bool}, and eq_{string} be the equality functions for the primitive types:

$$\begin{array}{rcll} H^{T(\overline{\alpha})}[\overline{f}, g, int] & = & eq_{int} & (7) \\ H^{T(\overline{\alpha})}[\overline{f}, g, string] & = & eq_{string} & (8) \\ H^{T(\overline{\alpha})}[\overline{f}, g, bool] & = & eq_{bool} & (9) \\ H^{T(\overline{\alpha})}[\overline{f}, g, \alpha_k] & = & f_{\alpha_k} & (10) \\ H^{T(\overline{\alpha})}[\overline{f}, g, T(\overline{\alpha})] & = & g & (11) \\ H^{T(\overline{\alpha})}[\overline{f}, g, S(t_1, \ldots, t_q)] & = & \mathrm{equal}^{S}\ (H^{T(\overline{\alpha})}[\overline{f}, g, t_1], \ldots, H^{T(\overline{\alpha})}[\overline{f}, g, t_q]) & (12) \\ H^{T(\overline{\alpha})}[\overline{f}, g, t_1 \times \ldots \times t_n] & = & (\tau_n^{\wedge}) \circ (H^{T(\overline{\alpha})}[\overline{f}, g, t_1] \times \ldots \times H^{T(\overline{\alpha})}[\overline{f}, g, t_n]) & (13) \\ & & & (14) \end{array}$$

where we extend the $\times$ notation to binary functions: $(f_1 \times \ldots \times f_n)((x_1, \ldots, x_n), (y_1, \ldots, y_n)) = (f_1(x_1, y_1), \ldots, f_n(x_n, y_n))$, and the distribution function, $\tau_n^{\wedge}$ is defined by $\tau_n^{\wedge}(x_1, \ldots, x_n) = x_1 \wedge \ldots \wedge x_n$.

We have used templates to define text based printing functions, and binary-format input and output of arbitrary (function free) data structures. In addition we have used *binary* templates and type parametric combinators to define generalized *zip* and *unification* functions as well.

7 Mutually Recursive Types

Type parametric combinators and templates may be extended to mutually recursive types in a straight-forward manner. Consider the mutually recursive types *exp* and *dec* below. These types might be used to represent expressions and declarations in a simple ML like language with only variables, applications, and let expressions:

$$\begin{array}{ll} & exp(\alpha) = Var(\alpha) \mid Let(dec(\alpha) \times exp(\alpha)) \mid Apply(exp(\alpha) \times exp(\alpha)) \\ \wedge & \\ & dec(\alpha) = Val(\alpha \times exp(\alpha)) \mid Fn(string \times \alpha \times exp(\alpha)) \end{array}$$

The free type variable α represents the carrier type for variables, for example it might be `string`. A type parametric combinator for a set of n mutually recursive types, $T_1(\alpha_1, \ldots, \alpha_p)$, $\ldots$, $T_n(\alpha_1, \ldots, \alpha_p)$ each with p type variables will need to "combine" $n + p$ functions. In this section we will represent these functions as $\overline{f}, \overline{g}$, which stand for $f_1, \ldots, f_p$ and $g_1, \ldots, g_n$. Thus to extend the combinator M of Definition 2 we need only modify Equation 2:

$$M^{T_1(\overline{\alpha}),\ldots,T_n(\overline{\alpha})}[\overline{f}, \overline{g}, T_i(\overline{\alpha})] \quad = \quad g_i$$

Invocation of a template maps the template equations over all constructors of each of the mutually recursive types. Thus the ordinary map for exp and dec is:

$$\begin{array}{llcl} & map^{exp}\ (f)\ (Var\ x) & = & Var(f\ x) \\ & map^{exp}\ (f)\ (Let(d, e)) & = & Let(map^{dec}\ (f)\ d,\ map^{exp}\ (f)\ e) \\ & map^{exp}\ (f)\ (Apply(g, e)) & = & Apply(map^{exp}\ (f)\ g,\ map^{exp}\ (f)\ e) \\ \wedge & & & \\ & map^{dec}\ (f)\ (Val(s, e)) & = & Val(s,\ map^{exp}\ (f)\ e) \\ & map^{dec}\ (f)\ (Fn(s, x, e)) & = & Fn(s, f\ x,\ map^{exp}\ (f)\ e) \end{array}$$

8 Compile-time Reflection

Type parametric combinators and templates have been implemented in the compile-time reflective subset of ML we call CRML (Compile-time Reflective ML).

Language tools usually consist of an *object language* in which the programs that are being manipulated are expressed, and a *meta language* that is used to describe the manipulation. A compile time reflective language has features that allow it to be its own meta-language. In CRML the object language is "encoded" (represented) in an ML datatype. There is a datatype for each syntactic feature of ML. Object language manipulations are described by manipulations of this "representation" datatype. CRML contains syntactic sugar (object brackets `<< >>`, and escape `‘`) for constructing and pattern matching program representations that *mirror the corresponding actual programs.* Thus, meta programs manipulating object programs may either be expressed directly with the explicit constructors of the representation type or with this "object-language" extension to ML's syntax. Text within the object-language brackets (`<< >>`) is parsed but not compiled. Its representation is returned as the value. Meta-language expressions may be included in the object-language text by "escaping" them with a backquote character (`‘`). Samples of this feature are illustrated in the table below:

Concrete syntax	Constructor based	Object bracket based
`x`	`Id "x"`	`<< x >>`
`f x`	`App(Id "f",Id "x")`	`<< f x >>`
	`App(g,y)`	`` << `g `y >> ``
`(x,y)`	`Tuple [ Id "x", Id "y" ]`	`<< (x,y) >>`
	`Tuple [x, y]`	`` << `x * `y >> ``

By using reflection, generic operators, such as *map* and *fold*, have straight-forward implementations by computing over the representations of datatype declarations. In CRML a *template* defines a function which, when invoked, is mapped over all the constructors (and their corresponding types) of a datatype declaration, constructing the object language value for the representation of a function declaration. For example the template below defines a function `Gen_map` that generates the representation of a function declaration from a string (representing the name of a type constructor).

```
fun template Gen_map T =
    map f ((Ci of d -> r) xbar) = `Ci (`(M r <<f>> <<map f>> d) `xbar);
```

The expression in the constructor position of the function definition, `((Ci of d -> r) xbar)`, is treated as a pattern. Thus upon invocation of the template the variables in this pattern will be bound to object language values particular to each constructor. `Ci` is bound to an object language expression for the constructor function, `xbar` to an object language tuple expression (of the appropriate "shape" to be `Ci`'s argument), `d` to the object language type of `Ci`'s domain, and `r` to the object language type of `Ci`'s range (which is the type `T`).

The rest of the expression is taken "literally" to generate one of the equations defining a function, except that escaped expressions are evaluated at invocation time and "spliced" into the equation. In this template `M` is an ML version of the combinator M dicussed in the previous sections, except it computes over the representations of types and expressions.

While an escape character inside object brackets or a template definition allows the results of meta computations to be "spliced" into object programs, an unbracketed, escaped expression is a simple interface to compile-time reflection. It indicates that the escaped expression should be evaluated (at compile-time) to compute the expression (or type, pattern, declaration, etc.) that replaces the escaped expression (much like macro expansion).

Thus, using the `Gen_map` meta program, defined above, the program below calculates and defines the map for list:

```
val maplist = let `(Gen_map "list") in map end;
```

as if the user had typed the following instead:

```
val maplist = let fun map f Nil = Nil
                    | map f (Cons(a1,a2)) = Cons(f a1,map f a2)
              in  map end;
```

In this section we have given a taste of how compile-time reflection is used to provide type parametric programming capabilites. We have not attempted to give any formal semantics to CRML. A semantics for compile-time reflection can be found elsewhere [7].

8.1 Single use Combinators

In the this section we give an illustration of the use of CRML to solve a problem which illustrates an additional use for type parametric combinators beyond the generation of functions for a large class of type constructors. When a type has many constructors it is often easier to construct a function for such a type using a template than to code it by hand.

For example consider the type below which could encode the abstract syntax for a subset of expressions in ML.

```
datatype  'a exp =
     Var of 'a                                (* x             *)
|    Intconst of int                          (* 5             *)
|    Boolconst of bool                        (* true          *)
|    Stringconst of string                    (* "a"           *)
|    Apply of 'a exp * 'a exp                 (* f x           *)
|    Prod of ( 'a exp) list                   (* (1,"z")       *)
|    Fn of (pat * 'a exp) list                (* fn x => x+1   *)
|    Case of 'a exp * (pat * 'a exp) list;    (* case x of ... *)
```

A function which computes a list of variables which appears free in an `exp` could be code by hand or by using a template. Here we construct a combinator which is intended to be used only once, for this particular function. It describes what happens at each "point", and lets the template and type parametric combinator provide the plumbing which "connects" the points together.

For this example we assume the existence of a function `bound_in_pat` which returns a list of variables bound by a pattern. This function could be generated by a template as well but choose to assume it to keep the size of the example manageable.

```
fun bind pat xlist =
listdiff (bound_in_pat pat) xlist;
```

Thus if `xlist` is a list of variable free in `x`, and `x` appears in case "guarded" by the pattern `p` the function `bind` computes those variables free in that clause of the case.

The type parametric combinator `tpc` computes a function for each type that appears in the definition of `exp`. The template `Free` use the combinator to construct a function.

```
fun tpc t f = case t of
  <t<'a>> => <<fn x => [x]>>
| <t<'a exp>> => f
| <t<pat * 'a exp>> => <<fn (p,e) => bind p (`f e)>>
| <t<`t list>> => <<concatl o (map `(tpc t f))>>
| <t<`t * `s>> => <<fn (x,y) => (`(tpc t f) x) @ (`(tpc s f) y)>>
| _ => <<fn _ => []>>;


fun template Free T =
  free ((C of d -> r) X) = `(tpc d <<free>>) `X;
```

The function constructed by `‘(Free "exp")` follows:

```
fun free (Var a1) = [a1]
|   free (Intconst a1) = []
|   free (Boolconst a1) = []
|   free (Stringconst a1) = []
|   free (Apply (a1,a2)) = (free a1 @ free a2)
|   free (Prod a1) = concatl (map free a1)
|   free (Fn a1) = concatl (map (fn (p,e) => patbound p (free e)) a1)
|   free (Case2 (a1,a2)) = (free a1 @ concatl (map (fn (b1,b2) => patbound b1 (free b2)) a2
```

The template `Free` connects all the "points" to construct a complete function. For larger types this can be quite advantageous. We have used such techniques extensively in our implementation of CRML.

9 Conclusion

Abstraction is the key to functional programming. Abstraction over type constructors leads to style of programming that can be quite effective. Programming language designers have known this for a long time, and ad-hoc, limited methods of providing this functionality have appeared in a number of languages. Type parametric programming provides a user extendible, unifying mechanism to supplant these ad-hoc methods. In addition type parametric combinators provide a high level mechanism to prove properties about all functions generated by a particlar template.

Our experience in using these mechanisms to bootstrap our CRML system supports these contentions, and we imagine a much more limited version of reflection could be used to implement these ideas effectively in other more traditional languages.

10 Acknowledgements

The author would like to thank *Category Dick* Kieburtz and *Type-Safe Jim* Hook for many long conversations which help clarify these ideas.

References

[1] J. Cockett and D. Spencer. Strong Categorical Datatypes I. In R. Seely, editor, *International Meeting on Category Theory 1991*, Canadian Mathematical Society Proceedings, Vol. 13, pp 141-169. AMS, Montreal, 1992.

[2] H. Dybkjær. Category Theory, Types, and Programming Languages. Ph.D. thesis, Department of Computer Science, University of Copenhagen (DIKU), May 1991.

[3] M.M. Fokkinga Law and Order in Algorithmics Ph.D. thesis, University of Twente, Dept. INF, Enschede, The Netherlands. 1992.

[4] M.M. Fokkinga Calculate Categorically! Formal Aspects of Computing(1992) Vol 4, pp 673-692.

[5] T. Hagino. *A Categorical Programming Language.* Ph.D. thesis, University of Edinburgh, 1987.

[6] J. Hook, R. Kieburtz, and T. Sheard. Generating Programs by Reflection. Oregon Graduate Institute Technical Report 92-015.

[7] J. Hook and T. Sheard. A Semantics of Compile-time Reflection. Oregon Graduate Institute Technical Report 93-019.

[8] P. Hudak, S.P. Jones, P. Wadler, et.al. Report on the Programming Language Haskell. ACM Sigplan Notices, Volume 27, No. 5, May 1992

[9] E. Meijer, M. Fokkinga, and R. Paterson. Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire. In *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture, Cambridge, Massachusetts*, pp 124–144, August 1991.

[10] Ross Paterson. Control Structures from Types, 1993. Unpublished paper. Imperial College and Queen Mary and Westfield College, University of London, London

[11] T. Sheard, and L. Fegaras. A Fold for All Seasons. In *Conference on Functional Programming Languages and Computer Architecture*, Copenhagen, June 1993. pp 233-242

[12] P. Wadler and S. Blott. How to make *ad hoc* polymorphism less *ad hoc*. In *Proceedings of the 16th ACM Symposium on Principles of Programming Languages*. pp 60-76, Austin, Texas, Jan. 1989

[13] G. C. Wraith. A note on categorical datatypes. In D. H. Pitt, D. E. Rydeheard, P. Dybjer, A. M. Pits and A. Poigné, editors, *Category Theory and Computer Science*, volume 389 of *Lecture Notes in Computer Science*, pp 118-127. Springer Verlag, 1989.