FUNCTIONAL PEARLS

Programming with Recursion Schemes

Daniel C. Wang
Agere Systems
New Jersey, U.S.A.
(*e-mail:* dcwang@agere.com)

Tom Murphy VII
Carnegie Mellon
Pittsburgh, U.S.A.
(*e-mail:* tom7@cs.cmu.edu)

Abstract

Many typed functional languages provide excellent support for defining and manipulating concrete instances of inductively defined recursive types. However, few of these languages provide good support for treating these types in a more abstract way. There have been a number of language extensions proposed to provide abstract facilities for manipulating these types. Unfortunately none have been widely adopted. We show several programming idioms based on *recursion schemas* that provides many of the benefits of several proposed extensions without any needed language extensions.

By using recursion schemas we can completely hide the representation of an algebraic type while still fully supporting pattern matching with patterns of arbitrary depth, similar in many ways to Wadler's views (Wadler, 1987; Okasaki, 1998). We also can simulate or encode many features of refinement types (Freeman & Pfenning, 1991; Davies, 1997), polytypic recursion operators (Meijer *et al.*, 1991), and declarative rewriting.

Our programming idiom is more cumbersome in some areas when compared to the various proposed language extensions but provides a good deal of "bang for the buck." Our experience demonstrates that these techniques are especially useful for developing compilers which manipulate various specialized forms of lambda terms. Programming with recursion schemas gain many of the benefits of views, refinement types, polytypic programming, and declarative rewriting without the need for any extensions to the underlying language.

1 Introduction

Programming languages that support algebraic data types and pattern matching provide excellent support for manipulating concrete data structures. However, pattern matching typically requires the full representation of the data type to be exposed to both the programmer and compiler. This disallows data abstraction. Programmers are burdened with deciding between pattern matching syntax or data abstraction. Wadler has suggested an extension (Wadler, 1987) that avoids this tension. There also have been numerous proposed extensions for pattern match-

```
signature NAT = sig
    type t
    val zero     : t
    val succ     : t -> t
    val case_nat : t -> {zero:unit -> 'a, succ:t ->'a} ->'a
  end
```

Fig. 1. First attempt at an abstract interface.

ing to achieve similar effects (Aitken & Reppy, 1992; Fähndrich & Boyland, 1997; Erwig, 1997).

By using recursion schemas we can completely hide the representation of an algebraic type and still support pattern matching with patterns of arbitrary depth, similar in many ways to Wadler's views (Wadler, 1987; Okasaki, 1998). We can define abstract interfaces that simulate some of the features of refinement types (Freeman & Pfenning, 1991; Davies, 1997). In particular we can define explicit subtypes of a more general type and provide for zero cost coercions between a subtype and its supertype. We also can define a parameterized library that provides polytypic recursion operators (Meijer *et al.*, 1991) for a large class of interesting types. Using our polytypic recursion operators we show a declarative rewriting idiom. We provide numerous examples of using our approach as well as some simple benchmarks to evaluate how our idiom affects the efficiency of programs across several different ML compilers. We also report our experience in applying a simplified version of our technique to the TILT compiler (Petersen *et al.*, 2000; Murphy VII, 2002).

Using recursion schemas provides many benefits of several proposed language extensions by defining the appropriate abstract interface to an inductively defined algebraic type. For the purposes of discussion we will use Standard ML to present our examples, but our ideas easily extend to other languages that support algebraic data types and polymorphism.

To begin we will consider how to implement the following program so that the inductively define type `nat` remains abstract

```
datatype nat = Zero | Succ of nat

fun add (Zero, m) = m
  | add (n, Zero) = n
  | add (Succ n, Succ m) = Succ(Succ(add(n, m)))
```

1.1 The Obvious Interface

We wish to make the representation of the `nat` type abstract. The first interface that comes to mind is the one in Figure 1. Unfortunately this interface is cumbersome. We can see why when we rewrite `add` to use the interface

```
functor Add(structure N : NAT) = struct
```

```
    open N
    fun add (n,m) =
         case_nat n
         {zero=fn () => m,
          succ=fn n  => case_nat m
                        {zero=fn () => n,
                         succ=fn m => succ(succ(add(n, m)))}}
  end
```

We lose the benefits of pattern matching and are stuck programming with only primitive case constructs. Unless we perform some relatively aggressive optimizations we will be creating as many as four function closures in order to use this interface. So not only is this interface cumbersome it also potentially introduces ineffencies.

However, since the type that represents naturals is abstract we can choose either of these two implementations as a parameter to our module

```
structure SimpleNat : NAT = struct
    datatype t = Zero | Succ of t
    val zero = Zero
    val succ = Succ
    fun case_nat Zero     {zero, succ} = zero()
      | case_nat (Succ n) {zero, succ} = succ n
  end

structure FastNat : NAT = struct
    type t = int
    val zero = 0
    fun succ n = n + 1
    fun case_nat 0 {zero, succ} = zero()
      | case_nat n {zero, succ} = succ(n-1)
  end
```

We would like the freedom to choose between different representations without losing the benefits of pattern matching. This problem is quite apparent in the implementations of typed intermediate languages. To avoid exponential blowup type based compilers often apply hash-consing, memoization, explicit substitution, and lazy normalization to the representation of types (Shao, 1998). To hide all these representation techniques some compilers use a clumsy abstract interface. Figure 2 is an example of such an interface.

1.2 Partial Abstraction

Another interface approach is to only reveal the top-level structure of the the algebraic type and require the use of functions that coerce the concrete top-level structure into an abstract type. This *partial abstraction* approach is similar to compilation techniques used to avoid expensive boxing and unboxing coercions when

```
(* FLINT tkind is roughly equivalent to the
 * following ML datatype
 *  datatype tkind
 *      = TK_MONO
 *      | TK_BOX
 *      | TK_SEQ of tkind list
 *      | TK_FUN of tkind list * tkind
 *)

(* constructors *)
val tkc_mono : tkind
val tkc_box  : tkind
val tkc_seq  : tkind list -> tkind
val tkc_fun  : tkind list * tkind -> tkind

(* deconstructors *)
val tkd_mono : tkind -> unit
val tkd_box  : tkind -> unit
val tkd_seq  : tkind -> tkind list
val tkd_fun  : tkind -> tkind list * tkind

(* predicates *)
val tkp_mono : tkind -> bool
val tkp_box  : tkind -> bool
val tkp_seq  : tkind -> bool
val tkp_fun  : tkind -> bool

(* one-arm switch *)
val tkw_mono : tkind * (unit -> 'a) * (tkind -> 'a) -> 'a
val tkw_box  : tkind * (unit -> 'a) * (tkind -> 'a) -> 'a
val tkw_seq  : tkind * (tkind list -> 'a) * (tkind -> 'a) -> 'a
val tkw_fun  : tkind * (tkind list * tkind -> 'a) * (tkind -> 'a) -> 'a
```

Fig. 2. Abstract interface found in a type based compiler

.

compiling polymorphic values (Shao, 1997). This partial abstraction technique has been used as an interface in an experimental version of the TILT compiler (Murphy VII, 2002).

Figure 3 contains an interface for natural numbers that uses partial abstraction. Figure 3 replaces several functions with a concrete top-level type (`s`) and two functions (`inj` and `prj`). Notice that the type `s` is structurally the same as the original `datatype nat` definition except all the recursive occurrences of the type `nat` are replaced by the type `t`. As a convention we use all uppercase constructor names for the concrete top-level type that hides an abstract type to avoid confusing them with normal constructors from inductively defined types. The `inj` and `prj` functions are similar to the standard coercion operators seen in isorecursive (Pierce, 2002) for-

```
signature NAT = sig
    type t
    datatype s = ZERO | SUCC of t
    val inj : s -> t
    val prj : t -> s
  end
```

Fig. 3. Partially abstract interface.

mulations of recursive types. It requires a little work to get used to inserting them in programs but this becomes second nature with practice.

The following program is our `Add` example reworked with the partially abstract interface in Figure 3

```
functor Add(structure N : NAT) = struct
    open N
    fun add (n, m) =
      case (prj n, prj m) of
        (ZERO, m) => inj m
      | (n, ZERO) => inj n
      | (SUCC n, SUCC m) => inj(SUCC(inj(SUCC(add(n, m)))))
  end
```

Since our language is impure there is nothing preventing either `inj` or `prj` from causing arbitrary side effects. With explicit coercions the location of any possible effects are clear. The issue of when coercion related effects occur requires some thought when when adapting Wadler's proposal for *views* (Wadler, 1987) to eager languages such as ML (Okasaki, 1998). Regardless of these issues we can now use the standard pattern matching primitives to match against the constructors declared by `s`.

Below are two modules that implement the new `NAT` interface

```
structure SimpleNat : NAT = struct
    datatype s = ZERO | SUCC of s
    type t = s
    fun inj x = x
    fun prj x = x
  end

structure FastNat : NAT = struct
    type t = int
    datatype s = ZERO | SUCC of t
    fun inj ZERO = 0
      | inj (SUCC n) = n + 1
    fun prj 0 = ZERO
      | prj n = SUCC (n - 1)
```

```
  end
```

The partially-abstract interface in Figure 3 is quite useful but not as widely used as it could be. However, it still is not as general as we would like. Consider the program below which uses nested patterns.

```
val one = Succ(Zero)
fun fib Zero = one
  | fib (Succ Zero) = one
  | fib (Succ (Succ n)) = add(fib n,fib (Succ n))
```

Using our partially abstract interface requires us to rewrite our program as

```
functor Fib(structure N : NAT
            val add : N.t * N.t -> N.t) = struct
    open N
    val one = inj (SUCC (inj ZERO))
    fun fib n =
         case prj n of
           ZERO   => one
         | SUCC n => (case prj n of
                        ZERO   => one
                      | SUCC n => add (fib n, fib(inj(SUCC n))))
  end
```

This is unfortunate, since we are stuck programming with primitive case statements. We can generalize the partially abstract approach to allow us to program with arbitrarily nested patterns, as well as providing the benefits of refined types, generic recursion operators, and declarative rewriting.

2 Using Recursion Schemes

The key idea of our technique relies on the well known metatheory for algebraic types. Meijer, Fokkinga, and Paterson (Meijer *et al.*, 1991) present a good summary of the ideas. Their metatheory is a useful guide to understanding the generality of our technique. However, we wish to emphasize programming pragmatics and not metatheory. So we will continue to develop our technique through several illustrative examples, without directly referring to the metatheory.

Let us rework the previous interface `NAT` using a more general approach. Figure 4 contains a `NAT` interface that generalizes our previous interface that used partial abstraction. In Figure 4 we have simply replaced the type `s` with a polymorphic type constructor. The type constructor `'a F` is the same as the original `datatype s` definition except all the occurrences of the type `s` are replaced by the type variable `'a`. The type constructor `'a F` is a recursion schema that characterizes the pattern of recursion.

We modify the types of the `inj` and `prj` functions appropriately. Below are two modules that implement the new `NAT` interface

```
signature NAT = sig
    type t
    datatype 'a F = ZERO | SUCC of 'a
    val inj : t F -> t
    val prj : t -> t F
  end
```

Fig. 4. Interface using recursion schemas.

```
structure SimpleNat : NAT = struct
    datatype 'a F = ZERO | SUCC of 'a
    datatype t = Fix of t F
    fun inj x = Fix x
    fun prj (Fix x) = x
  end

structure FastNat : NAT = struct
    datatype 'a F = ZERO | SUCC of 'a
    type t = int
    fun inj ZERO     = 0
      | inj (SUCC n) = n + 1
    fun prj 0        = ZERO
      | prj n        = (SUCC (n - 1))
  end
```

Unlike our previous interface the `inj` and `prj` functions for `SimpleNat` are non-trivial and must wrap and unwrap an extra constructor we have introduced to tie the recursive knot. A optimizing compiler should be able to use the same representation as the more obvious concrete representation. The representation we have chosen for naturals in `SimpleNat` is a *two-level type*, since we have split the inductively defined type into two distinct levels. The first `'a F` represents the structure while `t` ties the recursive knot. Two-level types have been used in other situations to abstract the structure of inductively defined types in order to build generic algorithms that are independent of the structure (Sheard, 2001).

2.1 Handling Nested Patterns

We can handle nested patterns by defining two helper functions `inj2` and `prj2`. These work just like our `inj` and `prj` operators except they work with patterns that are two deep. The type of `inj2` is `t F F -> t`. The type of `prj2` is `t -> t F F`.

```
functor Fib(structure N : NAT
            val add : N.t * N.t -> N.t) = struct
    open N
    fun inj2 n = inj (case n of
```

```
                      ZERO   => ZERO
                    | SUCC n => SUCC(inj n))
  fun prj2 n = case prj n of
                 ZERO   => ZERO
               | SUCC n => SUCC(prj n)

  val one = inj2 (SUCC ZERO)
  fun fib n = case prj2 n of
                ZERO => one
              | SUCC ZERO     => one
              | SUCC (SUCC n) => add (fib n, fib (inj (SUCC n)))
end
```

We can easily predefine a family of `inj`n and `prj`n functions for as deep as necessary by defining the following utility modules

```
functor Injs(structure N : NAT) = struct
  open N
    fun inj_succ inj_pred n = inj (case n of
                                     ZERO   => ZERO
                                   | SUCC n => SUCC(inj_pred n))
    val inj1 = inj
    val inj2 = inj_succ inj1
    val inj3 = inj_succ inj2
    val inj4 = inj_succ inj3
    val inj5 = inj_succ inj4
  end

functor Prjs(structure N : NAT) = struct
  open N
    fun prj_succ prj_pred n = case prj n of
                                ZERO   => ZERO
                              | SUCC n => SUCC(prj_pred n)
    val prj1 = prj
    val prj2 = prj_succ prj1
    val prj3 = prj_succ prj2
    val prj4 = prj_succ prj3
    val prj5 = prj_succ prj4
  end
```

Notice how the `inj_succ` and `prj_succ` functions build functions that work at a deeper level by reusing functions that work on a shallower level. Later we will show how to define `prj_succ` and `inj_succ` in a generic way that will allow us to build `Injs` and `Prjs` modules that work for a large family of types.

Also note that our `prj_`n will eagerly apply coercions that may be necessary. Wadler's original view proposal lazily applies the needed coercions. If programmers

want to lazily apply coercions they must transform the nested matches in to a sequence of primitive matches. We consider this a deficiency of our approach.

2.2 Avoiding Representation Coercions

Now that we have a reasonable interface that allows us to use pattern matching syntax while hiding the representation, we can use our new abstraction facilities to enforce static invariants without paying expensive penalties.

Suppose that we wish to distinguish between odd and even natural numbers. The following definitions are what we would be forced to use without data abstraction techniques

```
datatype nat  = Zero | Succ of nat
     and even = EvenZero | EvenSucc of odd
     and odd  = OddSucc of even


fun evenToNat EvenZero = Zero
  | evenToNat (EvenSucc odd) = Succ(oddToNat odd)
and oddToNat (OddSucc even) = evenToNat even
```

Refinement types (Freeman & Pfenning, 1991; Davies, 1997) provide a convenient extension to the type system of ML that allows us to express the relationships above in a more abstract way

```
datatype nat  = Zero | Succ of nat
datasort even = Zero | Succ of odd
     and odd  = Succ of even
```

Here the `datasort` declaration introduces new constraints that the type checker uses to verify that the values declared as `odd` or `even` hold to the invariants imposed by the data refinement. Refinement-type systems have powerful type inference systems as well as extending the normal type systems with intersection types to give the programmer a very rich set of constraints they can specify and enforce. Unfortunately, refinement types have not been widely adopted in any programming language.

Refinement types avoid the need for representation coercion functions such as `evenToNat` and `oddToNat` which traverse the entire structure to coerce values of type `even` and `odd` to values of type `nat`. In a refinement-type system values with the sort `even` and `odd` are simply `nat` values that the compiler has statically verified to conform to extra sort constraints placed on it by the `datasort` declaration.

Rather than relying on refinement types we can express many of the constraints with the interface in Figure 5. Here is a generic implementation of the interface that is parameterized by the underlying type of natural numbers

```
functor RefineNat(structure N : NAT) : NATS = struct
    open N
    type nat = N.t
```

```
signature NATS = sig
    include NAT
    type nat = t
    type even
    type odd

    structure Even : sig
      datatype 'a F = ZERO | SUCC of 'a

      val inj   : odd F -> even
      val prj   : even -> odd F
      val toNat : even -> nat
    end

    structure Odd : sig
      datatype 'a F = SUCC of 'a

      val inj   : even F -> odd
      val prj   : odd -> even F
      val toNat : odd -> nat
    end

    val addOddOdd   : (odd  * odd)  -> even
    val addEvenEven : (even * even) -> even
    val addOddEven  : (odd  * even) -> odd
    val addEvenOdd  : (even * odd)  -> odd
  end
```

Fig. 5. An interface for both odd and even natural numbers.

```
    type even = N.t
    type odd = N.t

    structure Even = struct
        datatype 'a F = ZERO | SUCC of 'a

        fun inj ZERO     = N.inj (N.ZERO)
          | inj (SUCC n) = N.inj (N.SUCC n)
        fun prj n = case N.prj n of
                      N.ZERO   => ZERO
                    | N.SUCC n => SUCC n
        fun toNat n = n
      end

    structure Odd = struct
        datatype 'a F = SUCC of 'a
```

```
      fun inj (SUCC n) = N.inj (N.SUCC n)
      fun prj n = case N.prj n of
                    N.ZERO   => raise (Fail "impossible")
                  | N.SUCC n => SUCC n
      fun toNat n = n
    end

  fun add (n, m) =
       case (prj n, prj m) of
         (ZERO, m) => inj m
       | (n, ZERO) => inj n
       | (SUCC n, SUCC m) => inj(SUCC(inj(SUCC(add(n, m)))))

  val addOddOdd   = add
  val addEvenEven = add
  val addOddEven  = add
  val addEvenOdd  = add
end
```

Notice that the `toNat` functions are the identity. Unlike a refinement-type system we have to trust the correctness of the code and the interface above. In particular the properties of our four different variants of the addition function are assumed to be correct. Systems with refinement types are able to verify that our addition function actually has the properties enumerated in the interface. However, a bug in our reasoning or code will only result in an unexpected exception being raised and not compromise the type safety of the system. Also, if an unexpected exception is raised the abstraction guarantees help us isolate the bug to a particular module.

At first glance this extra flexibility may seem a bit needless, but in practice this style of programming is especially useful for dealing with various lambda calculi used in compilers for functional programming languages. Typically the compiler deals with many refinements of an underlying calculus. This is done by duplicating types and constructors. It also requires that we duplicate a great deal of infrastructure which should be reusable in an ideal world. We provide a detailed example of representing ANF and CPS terms uniformly as general lambda terms as an Appendix.

2.3 Recursion Operators

It is quite useful to define generic recursion operators for our type. Figure 6 are typical interfaces for the recursion operators `fold` and `unfold`. Below is a simple module that demonstrates the use of the operators in Figure 6.

```
functor Convert(structure Rec : NAT_REC) : sig
  structure N : NAT
   val toInt   : N.t -> int
   val fromInt : int -> N.t
```

```
signature NAT_REC = sig
    structure N : NAT
    val fold : {succ:'a -> 'a, zero:unit -> 'a} -> N.t -> 'a

    datatype ('a,'b) sum = L of 'a | R of 'b
    type 'a gen = 'a -> ('a, unit) sum
    val unfold : 'a gen -> 'a -> N.t
  end
```

Fig. 6. Traditional recursion operators for natural numbers.

```
  end = struct
   open Rec
   fun toInt n = fold {zero=fn () => 0, succ=fn i => i + 1} n
   fun fromInt i =
     if i < 0 then raise (Fail "not a nat")
     else unfold (fn 0 => R () | n => L (i - 1)) i
  end
```

The function `toInt` defines a function that inductively deconstructs natural numbers and builds an integer representation of the number. The function `fromInt` defines a function that inductively constructs a natural number from an integer representation of the number. Below is a straightforward implementation of the operators `fold` and `unfold`.

```
functor NatRec(structure N : NAT) : NAT_REC =  struct
    structure N = N
    open N
    fun fold {succ, zero} n = let
      fun f ZERO     = zero ()
        | f (SUCC n) = succ (f (prj n))
      in f (prj n)
      end

    datatype ('a,'b) sum = L of 'a | R of 'b
    type 'a gen = 'a -> ('a, unit) sum
    fun unfold gen init = let
      fun g (R ()) = ZERO
        | g (L x)  = SUCC (inj (g (gen x)))
      in inj (g (gen init))
      end
  end
```

Unfortunately the interface in Figure 6 is a little confusing. The interface to `unfold` in particular is confusing to someone not already familiar with unfold. The

```
signature NAT_REC = sig
    structure N : NAT
    val fold   : ('a N.F -> 'a) -> N.t -> 'a
    val unfold : ('a -> 'a N.F) -> 'a -> N.t
  end
```

Fig. 7. Using recursion schemas for recursions operators.

relationship between folds and unfolds is completely lost by this interface. Recursion schemas allow us to clean up the interfaces and implementations of the operators.

Figure 7 is an alternative definition that uses recursion schemas. Here is a use of the interface in Figure 7.

```
functor Convert(structure Rec : NAT_REC) : sig
  structure N : NAT
   val toInt   : N.t -> int
   val fromInt : int -> N.t
  end = struct
   open Rec
   fun toInt n = fold (fn N.ZERO => 0
                        | N.SUCC n => n + 1) n
   fun fromInt i =
     if i < 0 then raise (Fail "not a nat")
     else unfold (fn 0 => N.ZERO
                   | i => N.SUCC (i - 1)) i
  end
```

Here is an implementation of the recursion operators

```
functor NatRec(structure N : NAT) : NAT_REC =
  struct
    structure N = N
    open N
    fun fold step n = let
      fun f ZERO     = step ZERO
        | f (SUCC n) = step (SUCC (f (prj n)))
      in f (prj n)
      end
    fun unfold gen init = let
      fun g ZERO     = ZERO
        | g (SUCC x) = SUCC (inj (g (gen x)))
      in inj (g (gen init))
      end
  end
```

```
signature TYP = sig
    type t
    type 'a F
    val Fmap : ('a -> 'b) -> 'a F -> 'b F
    val inj  : t F -> t
    val prj  : t -> t F
  end
```

```
Fmap (fn x => x)   ≡   (fn x => x)
    Fmap (f o g)   ≡   (Fmap f) o (Fmap g)
```

Fig. 8. Categorical interface to recursive types.

Our new definitions for `fold` and `unfold` are uniform and simpler. These definitions have more than an esthetic value. Next we will show how to write an entire family of generic fold and unfold functions that are independent of the underlying structure of the type.

3 Polytypic Functions

The metatheory for defining `fold` and `unfold` for arbitrary recursive types begins by representing the type as a categorical functor. A categorical functor acts on both types and values. In particular the constructor `'a F` is a function from types to types and represents the type portion of the functor that defines the underlying recursive type. Once we flesh out our interface to include a function `Fmap` that acts on values, we will have fully specified a functor and can use the constructions in Meijer, Fokkinga, et al. (Meijer *et al.*, 1991) to specify `fold` and `unfold` in a generic way.

The interface in Figure 8 describes the typing constraints and semantic constraints need for our categorical representation of recursive types.

There are several semantic constraints that must be imposed on the various operations for the derivations in Meijer, Fokkinga, et al. to hold. One semantic constraint placed on `Fmap` is that it is "structure preserving," i.e. the equations in Figure 8 must hold. These implicit semantic constraints must be assumed to hold for our constructions to be sensible. Below is a generic interface for recursion operators based on the categorical interface.

```
signature REC = sig
    structure T : TYP
    val fold    : ('a T.F -> 'a) -> T.t -> 'a
    val unfold  : ('a -> 'a T.F) -> 'a -> T.t
  end
```

The definitions for `fold` and `unfold` operators based on our interface and the equations presented in Meijer et al. are as follows:

```
functor Rec(structure T : TYP) : REC = struct
```

```
    structure T = T
    open T
    fun wrapF f g h x = f (Fmap h (g x))
    fun fold step x =  wrapF step prj (fold step) x
    fun unfold gen y = wrapF inj gen (unfold gen) y
  end
```

Notice that we do not have to examine the concrete structure of our type to define the operators. The structure of our type is implicitly encoded in the behavior of `Fmap`. Here is a simple example usage of this generic module.

```
functor NatRec(structure N : NAT) : NAT_REC =
  struct
    structure N = N
    structure T =
      struct
        type t = N.t
        type 'a F = 'a N.F
        val inj = N.inj
        val prj = N.prj
        fun Fmap f N.ZERO = N.ZERO
          | Fmap f (N.SUCC x) = N.SUCC(f x)
      end
    structure Rec = Rec(structure T = T)
    open Rec
  end
```

Notice that the defined `Fmap` is a structure preserving function. This is a critical property needed for the construction to hold.

The `Fmap` function we've define has other important uses. We can use it to express `inj_succ` and `prj_succ` in terms of `Fmap` also. So our modules `Injs` and `Prjs` become generic.

```
functor Injs(structure T : TYP) = struct
    open T
    fun inj_succ inj_pred x = inj (Fmap inj_pred x)
    val inj1 = inj
    val inj2 = inj_succ inj1
    val inj3 = inj_succ inj2
    val inj4 = inj_succ inj3
    val inj5 = inj_succ inj4
  end

functor Prjs(structure T : TYP) =
  struct
    open T
    fun prj_succ prj_pred x = Fmap prj_pred (prj x)
```

```
    val prj1 = prj
    val prj2 = prj_succ prj1
    val prj3 = prj_succ prj2
    val prj4 = prj_succ prj3
    val prj5 = prj_succ prj4
  end
```

It is useful to collect all these functions into one reusable module with the following interface:

```
signature AUX_DEFS = sig
    structure T  : TYP
    val inj1     : T.t T.F -> T.t
    val inj2     : T.t T.F T.F -> T.t
    val inj3     : T.t T.F T.F T.F -> T.t
    val inj4     : T.t T.F T.F T.F T.F -> T.t
    val inj5     : T.t T.F T.F T.F T.F T.F -> T.t
    val inj_succ : ('a -> T.t) -> 'a T.F -> T.t

    val prj1     : T.t -> T.t T.F
    val prj2     : T.t -> T.t T.F T.F
    val prj3     : T.t -> T.t T.F T.F T.F
    val prj4     : T.t -> T.t T.F T.F T.F T.F
    val prj5     : T.t -> T.t T.F T.F T.F T.F T.F
    val prj_succ : (T.t -> 'a) -> T.t -> 'a T.F

    val fold     : ('a T.F -> 'a) -> T.t -> 'a
    val unfold   : ('a -> 'a T.F) -> 'a -> T.t
  end
```

The module itself is simple implemented as

```
functor AuxDefs(structure T : TYP) : AUX_DEFS = struct
    structure T = T
    structure Injs = Injs(structure T = T)
    structure Prjs = Prjs(structure T = T)
    structure Rec  = Rec(structure T = T)
    open Injs Prjs Rec
  end
```

This module is applicable to any monomorphic recursive type regardless of the structure. Extending our examples to handle polymorphic types is straightforward. For a language like ML that lacks higher-order polymorphism one needs a set of these definition for each type constructor of a different arity.

4 Declarative Rewriting

Recursion schemas also allow for a more declarative programming style. Given an interface for well formed formulas we can write a simplifier for it by specifying it as a set of atomic rewrites that are inductively applied to a formula in a top-down way.

First let us define our interface

```
signature WFF = sig
    type t
    datatype 'a F = FALSE
                  | TRUE
                  | VAR of string
                  | AND of ('a * 'a)
                  | OR  of ('a * 'a)
                  | NOT of 'a

    val Fmap : ('a -> 'b) -> 'a F -> 'b F
    val inj  : t F -> t
    val prj  : t -> t F
  end
```

Next we define a function to rewrite values that adhere to the interface

```
functor Simplify(structure Wff : WFF) : sig
  structure Wff : WFF
  val simplify  : Wff.t -> Wff.t
end = struct
  structure AuxDefs = AuxDefs(structure T = Wff)
  structure Wff = Wff
  open AuxDefs Wff
  fun simplify e = let
  fun rewrite (NOT FALSE)     = TRUE
    | rewrite (NOT TRUE)      = FALSE
    | rewrite (NOT(NOT x))    = prj1 x
    | rewrite (AND(TRUE, x))  = x
    | rewrite (AND(x, TRUE))  = x
    | rewrite (AND(FALSE, x)) = FALSE
    | rewrite (AND(x, FALSE)) = FALSE
    | rewrite (OR(x, TRUE))   = TRUE
    | rewrite (OR(TRUE, x))   = TRUE
    | rewrite (OR(FALSE, x))  = x
    | rewrite (OR(x, FALSE))  = x
    | rewrite (x)             = (Fmap inj1) x
  in unfold (rewrite o prj2) e
  end
end
```

The function `rewrite` operates on patterns of depth two. So we must insert the appropriate coercions to make the types work out. Notice that we are using `unfold` to achieve a top-down rewrite of the term. Compare the approach above to the more traditional top-down rewrite

```
structure SimplifyWff = struct
  datatype t = False
             | True
             | Var of string
             | And of (t * t)
             | Or  of (t * t)
             | Not of t

  fun simplify e = let
    fun f (Not False)     = True
      | f (Not True)      = False
      | f (Not(Not x))    = f x
      | f (And(True, x))  = f x
      | f (And(x, True))  = f x
      | f (And(False, x)) = False
      | f (And(x,False))  = False
      | f (Or(x,True))    = True
      | f (Or(True, x))   = True
      | f (Or(False, x))  = f x
      | f (Or(x,False))   = f x
      | f (Not x)         = Not(f x)
      | f (And(x,y))      = And(f x, f y)
      | f (Or(x,y))       = Or (f x, f y)
      | f x               = x
    in f e
    end
end
```

Notice that last clause of the function `rewrite` in our declarative approach must be expanded into four clauses in the traditional approach. Our declarative approach is less error prone and more concise than the more traditional explicitly recursive technique. However, it does take a little practice to learn where to insert the coercions.

5 Limitations

The technique presented so far easily handles all types constructed with sums and products. Extending it to the polymorphic case is straightforward. It is not clear if we can extend our approach to handle *exponential types* such as

```
datatype istream = Susp of unit -> (int * istream)
```

where there is a recursive occurrence of a type in a function type. However, there already exists well understood metatheory for this (Meijer & Hutton, 1995) which we should be able to adapt.

What we have presented is more than adequate for dealing with abstract syntax trees and other common types found in the implementation of compilers and language processing tools. This is especially true for tools that deal with lambda terms.

6 Performance Impact

We report some very basic numbers for the following programs to understand the performance impact of our programming idiom, and to highlight some performance issues for compiler writers. Below are four different implementations of `fib`. Each one uses a different representation of natural numbers and an different interface to the representation.

```
structure ProgramA = struct
    datatype nat = Zero | Succ of nat
    fun add (Zero, m) = m
      | add (n, Zero) = n
      | add (Succ n, Succ m) = Succ(Succ(add(n, m)))
    val one = Succ(Zero)
    fun fib Zero = one
      | fib (Succ Zero) = one
      | fib (Succ (Succ n)) = add(fib n, fib (Succ n))
  end

structure ProgramB =
  Fib(structure N = SimpleNat
       open N
        fun add (n, m) =
         case (prj n, prj m) of
           (ZERO, m) => inj m
         | (n, ZERO) => inj n
         | (SUCC n, SUCC m) => inj(SUCC(inj(SUCC(add(n, m))))))

structure ProgramC =
  struct
    fun fib 0 = 1
      | fib 1 = 1
      | fib n = (fib (n - 2)) + (fib (n - 1))
  end

structure ProgramD =
  Fib(structure N = FastNat
```

```
val add = (op +))
```

Programs A and B use the simple but naive inductively defined representation of naturals. Programs C and D use integers to represent naturals. Programs A and C directly access the representation of naturals while programs B and D use the same abstract interface to the different representations.

Table 1 contains the runtimes of the programs averaged over ten runs when computing `fib(24)`[1] on a 1Ghz x86 processor running Linux for a variety of ML compilers. Programs A and B are using what should be isomorphic representations

Table 1. *Wall clock runtime in milliseconds.*

	Simple		Fast	
Compiler	A	B	C	D
MLton 20020410	69	67	4	4
PolyML 4.1.2	92	99	4	23
SML/NJ 110.0.7	93	263	6	22
MLKit 4.0	96	236	4	13
SML/NJ 110.41	140	361	4	15
Moscow ML 2.0	525	884	15	93

of natural numbers. Program B however runs slower in some instances because the compiler does not remove the interface overheads or deal with the two-level representation for the natural type efficiently. Programs C and D are using the same fast representation of natural numbers, and program D is slower in some cases because of interface overheads. It is important to note that the MLton compiler smart is able to optimize away all the overhead associated with both our interfaces. Although the MLton compilers is a whole-program optimizing compiler. The optimizations need for this idiom should not require whole-program analysis. In particular cross module inlining, smart representation decisions, and some local constant folding should be sufficient. It also seems the newer versions of SML/NJ has made some representation decisions that are negatively impacting performance.

Since programs A and B are using the naive representation, they are significantly slower than programs C and D across all compilers. In all cases Program D which uses our abstract interface with our fast representation of naturals runs faster then programs A or B. Notice that program D was simply obtained by applying a different implementation of a module implementing the `NAT` interface while program C is a complete rewrite of program A. In short if a change in representation can provide significant algorithmic improvements or software engineering benefits the overhead of using an abstract interface may be worth it.

[1] We choose n = 24 because larger values caused some implementations to fail.

We hope that all compiler writers will consider adding optimizations that recognize this particular idiom of programming and further reduce the cost of using it.

7 Case Study: TILT

In order to address the issue of scaling this idiom to large programs, we give a case study of a significant real-world application, the TILT compiler for Standard ML.

TILT (Petersen *et al.*, 2000) is type-directed so that it transforms types along side code and uses type information to perform optimizations as well as to help verify the compiler's work and debug errors in the compiler. A naive representation of types may lead to exponential slowdowns in both space and time. TILT's intermediate stages, in particular, must manipulate a large amount of type information. Though TILT takes care to do so in an asymptotically efficient way, TILT does not employ strategies such as hash consing, de Bruijn indexing, and explicit substitutions (Shao, 1998) in order to exploit redundancy that can only be detected at run-time. As an experiment, we decided to test how the TILT compiler might benefit from these techniques.

TILT originally used a concrete type to represent its intermediate language. This poses a problem for experimenting with different representations, since using a concrete type commits to one representation. The intermediate phase of TILT consists of more than 25,000 lines of code that manipulate these types directly. Therefore, the idea of changing to a FLINT-style abstract interface (as in Section 1.1) was rather daunting, especially considering that we wanted to try different combinations of techniques, and weren't sure if we would be able to achieve any benefit at all.

As a concession to efficiency over generality we rewrote the TILT compiler to use a partially-abstract interface (as in Section 1.2), rather than using the fully general recursion schema approach. This allows us to reuse the existing TILT representation of types as the underlying abstract type just as the version of `SimpleNat` does in Section 1.2.

Retrofitting the existing code to use this new interface was a tedious but simple task. The most significant work comes from rewriting nested patterns. We believed that our `prj` and `inj` operations would be somewhat expensive, so we did not want to expose several levels at once, since this forces extra calls to `prj` when they may not be needed. Therefore, we needed to manually rewrite nested patterns as primitive case statements.

We discovered that nested patterns are actually quite rare. In the 10,000 lines of type checker and support code, there were only approximately ten patterns (out of hundreds) that needed non-trivial changes. Had we been writing the code from scratch, the lack of nested patterns would have posed no problem at all.

We found that the type checker incurred a 20% speed penalty for using the partially abstract interface. This seemed to be due to lack of cross-module optimizations (such as inlining) and the disabling of low-level optimizations for datatypes. In the end, it also turned out that the TILT compiler's existing approach for dealing with

types in an asymptotically efficient way could not be easily improved on by various optimized representations. However, being able to experiment with various different representations without having to continually remodify clients of the intermediate language was a definite win.

It would be very desirable for compilers to implement the appropriate optimizations so that 20% speed penalty for just using the interface could be reduced to zero. Further details of the various representation experiments carried out can be found in Murphy (Murphy VII, 2002).

8 Related Work

Wadler (Wadler, 1987) presents the first concrete proposal to provide data abstraction and pattern matching in a relatively clean way. However, extending some of his ideas for eager impure languages is not as straightforward as one would like (Okasaki, 1998). In particular to provide sensible semantics in impure languages the results of view coercions need to be memoized. Our idiom creates an explicit intermediate data structure, the recursion schema, that provides the effect of memoizing the view coercions.

Meijer, Fokkinga, and Paterson present a metatheory for recursive types and apply it to derive equational reasoning principals for generalized recursion operators such as `fold` and `unfold`. In a follow up paper (Meijer & Hutton, 1995) they reintroduce their metatheory as a set of definitions for the functional language Gopher. They also extend their metatheory to handle function spaces. However, they do not directly apply their metatheory to hide the underlying representation of the values.

There also have been numerous extensions proposed to extend the power of pattern matching (Aitken & Reppy, 1992; Fähndrich & Boyland, 1997). They provide some syntactic sugar and are typically compile time abstractions that have no associated runtime overhead. Refinement types also provide some similar benefits but can be used to enforce and track many more non-trivial static properties. Views provide most of the benefits of all these systems but may incur some runtime penalties, because view coercions have computational effect. Active patterns (Erwig, 1997) offer yet another type of syntactic sugar. We believe we can extend our framework to simulate Erwig's active patterns with only a small loss in syntactic sugar. Our approach of only revealing the top-level structure of an abstract type is similar to some compilation techniques used to avoid expensive coercion functions when compiling polymorphic values (Shao, 1997).

9 Conclusion

We have shown a simple and powerful programming idiom. Our idiom provides programmers with pattern matching interfaces to abstract values. The technique is a simple embedding of the well understood categorical metatheory of recursive types into a concrete language. The power and generality of the metatheory leads to simple and pleasant programming interfaces. These interfaces impose a certain

amount of runtime and programming overhead but the flexibility of being able to choose different representations or avoid representation coercions can pay for many of the associated overheads of our technique. Adding some sort of compiler or language support for our idiom would be very desirable.

Arguably programming with recursion schemas is not as pleasant as having direct language support for views, refinement types, polytypic languages, or numerous other possible language extensions. However, recursion schemas have the very pragmatic advantage of not relying on any non-standard language extensions. They are a practical and useful programming idiom that can be used in any existing language that provides higher-order functions, polymorphism, and pattern matching.

From a language design standpoint the fact we can encode many different features of various language extensions in pragmatic yet clumsy ways suggests that there maybe some simple extensions to existing languages that will buy us some of the power and expressiveness of many seemingly diverse extensions in a unified and elegant way. For example having the type checker attempt to automatically insert `inj` and `prj` functions would make programming with recursion schemas much more pleasant, and almost completely transparent to the programmer.

10 Acknowledgments

Thanks to Matthias Blume, Robert Harper, John Reppy, and Philp Wadler for reviewing earlier drafts and suggesting improvements, as well as the anonymous reviewers.

References

Aitken, William E., & Reppy, John H. 1992 (June). Abstract value constructors. *Pages 1–11 of: Proceedings of the 1992 ACM workshop on ML and its applications.*

Davies, Rowan. (1997). A refinement-type checker for Standard ML. *Pages 565–?? of: International conference on algebraic methodology and software technology.* Lecture Notes in Computer Science, vol. 1349. Springer-Verlag.

Erwig, Martin. (1997). Active patterns. *Lecture notes in computer science*, **1268**, 21–40.

Fähndrich, Manuel, & Boyland, John. 1997 (June). Statically checkable pattern abstractions. *Pages 75–84 of: Proceedings of the 1997 ACM SIGPLAN international conference on functional programming.*

Flanagan, Cormac, Sabry, Amr, Duba, Bruce F., & Felleisen, Matthias. (1993). The essence of compiling with continuations. *Pages 237–247 of: Proceedings acm sigplan 1993 conf. on programming language design and implementation, pldi'93, albuquerque, nm, usa, 23–25 june 1993.* SIGPLAN Notices, vol. 28(6). New York: ACM Press.

Freeman, Tim, & Pfenning, Frank. (1991). Refinement types for ML. *Pages 268–277 of: Proceedings of the SIGPLAN '91 symposium on language design and implementation.* Toronto, Ontario: ACM Press.

Meijer, Erik, & Hutton, Graham. (1995). Bananas in space: Extending fold and unfold to exponential types. *Pages 324–333 of: Proceedings of the seventh international conference on functional programming languages and computer architecture.* La Jolla, California: ACM Press, for ACM SIGPLAN/SIGARCH and IFIP WG2.8.

Meijer, Erik, Fokkinga, Maarten, & Paterson, Ross. (1991). Functional programming

with bananas, lenses, envelopes and barbed wire. *Pages 124–144 of:* Hughes, John (ed), *Proceedings of the fifth international conference on functional programming languages and computer architecture*. LNCS, vol. 523. Cambridge, MA, USA: Springer, for ACM.

Murphy VII, Tom. 2002 (Mar.). *The wizard of TILT: Efficient, convenient and abstract type representations*. Tech. rept. CMU-CS-02-120. School of Computer Science, Carnegie Mellon University.

Okasaki, Chris. 1998 (Sept.). Views for Standard ML. *Pages 14–23 of: Sigplam workshop on ml*.

Petersen, Leaf, Cheng, Perry, Harper, Robert, & Stone, Chris. 2000 (Mar.). *Implementing the TILT internal language*. Tech. rept. CMU-CS-00-180. School of Computer Science, Carnegie Mellon University.

Pierce, Benjamin C. (2002). *Types and programming languages*. MIT Press. Pages 275–278.

Shao, Zhong. 1997 (June). Flexible representation analysis. *Pages 85–98 of: Proceedings of the 1997 ACM SIGPLAN international conference on functional programming*.

Shao, Zhong. 1998 (Sept.). Implementing typed intermediate language. *Pages 313–323 of: Proceedings of the 1998 ACM SIGPLAN international conference on functional programming*.

Sheard, Tim. 2001 (Sept.). Generic unification via two-level types and parameterized modules. *Pages 86–97 of: Proceedings of the 2001 ACM SIGPLAN international conference on functional programming*.

Wadler, Philip. (1987). Views: A way for pattern matching to cohabit with data abstraction. *Pages 307–312 of: Proceedings, 14th symposium on principles of programming languages*. Association for Computing Machinery.

A A Basic Lambda Compiler

We want to provide a non-trivial example of our programming idiom by fleshing out the details of a compiler for an untyped lambda calculus. Each phase of the compiler will operate on a different refinement of lambda terms. We will only present some illustrative fragments of the complete system. Full source code can be obtained from *TODO insert url*.

A.1 Interfaces

We begin with a few modules that provide some basic types which are shared across each refinement of lambda terms.

```
structure Var :> sig
  type t
  val var      : string -> t
  val fresh    : string -> t
  ...
end = ...
structure Prim :> sig
  type t
  ...
end = ...
```

Here is the interface for general lambda terms using our technique of recursion schemas

```
signature LAM =
  sig
    type var  = Var.t
    type prim = Prim.t
    type t
    datatype 'a F =
      VAR  of var
    | PRIM of prim * 'a list
    | APP  of 'a * 'a list
    | LAM  of var list * 'a

    val Fmap : ('a -> 'b) -> 'a F -> 'b F
    val inj  : t F -> t
    val prj  : t -> t F
  end
```

Next we provide an interface for terms in a-normal form (ANF) (Flanagan *et al.*, 1993) which is a syntactic restriction of general lambda terms. ANF terms bind every non-trivial expression to a variable. ANF terms also make the order of evaluation and the difference between tail and non-tail calls explicit. Tail-calls are represented by the `JUMP` constructor.

```
signature ANF =
  sig
    type var  = Var.t
    type prim = Prim.t
    type t
    datatype 'a F =
      RET  of var
    | PRIM of var * (prim * var list) * 'a
    | FUN  of var * (var list * 'a) * 'a
    | CALL of var * (var * var list) * 'a
    | JUMP of var * var list

    val Fmap : ('a -> 'b) -> 'a F -> 'b F
    val inj  : t F -> t
    val prj  : t -> t F
  end
```

Finally we represent CPS terms. CPS terms can be seen as a refinement of ANF terms where all calls are tail-calls and no function returns.

```
signature CPS =
  sig
```

```
    type var  = Var.t
    type prim = Prim.t
    type t

    datatype 'a F =
      HALT of var
    | PRIM of var * (prim * var list) * 'a
    | FUN  of var * (var list * 'a) * 'a
    | JUMP of var * var list

    val Fmap : ('a -> 'b) -> 'a F -> 'b F
    val inj  : t F -> t
    val prj  : t -> t F
  end
```

An optimizing compiler will manipulate general terms as well as the ANF and CPS refinements. Below is an interface for a set of functions that are useful in the implementation and debugging of such a compiler.

```
signature COMPILE =
  sig
    structure Lam : LAM
    structure ANF : ANF
    structure CPS : CPS

    val lam_toString  : Lam.t -> string
    val lam_freeVars  : Lam.t -> Var.t list
    val lam_eq        : Lam.t -> Lam.t -> bool
    val lam_alpha_eq  : Lam.t -> Lam.t -> bool
    val lam_rename    : Lam.t -> Lam.t

    val anf_normalize : Lam.t -> ANF.t
    val anf_toString  : ANF.t -> string
    val anf_toLam     : ANF.t -> Lam.t
    val anf_rename    : ANF.t -> ANF.t

    val cps_convert   : ANF.t -> CPS.t
    val cps_toString  : CPS.t -> string
    val cps_toLam     : CPS.t -> Lam.t
    val cps_toANF     : CPS.t -> ANF.t
    val cps_rename    : CPS.t -> CPS.t
  end
```

The functions `anf_toLam` and `cps_toLam` allow us to reuse functions such as `lam_freeVars`, `lam_eq`, or `lam_alpha_eq` for ANF and CPS terms. Our interface provides for three separate functions to convert terms to strings for debugging

purposes as convenience, and to allow for the possibility of using specialized pretty printers for each type.

We would like an implementation of the `COMPILER` interface to make functions such as `anf_toLam` and `cps_toLam` be implemented as identity functions.

A.2 Implementations

We start with a simple implementation of the `LAM` interface. In the module below terms are represented with a concrete algebraic type and we simply provide some interface functions. A two-level representation would be more elegant but depending on the compiler may not be as efficient as this more heavy handed approach.

```
structure Lam : LAM =
  struct
    type var  = Var.t
    type prim = Prim.t
    datatype t =
      Var  of var
      ...

    datatype 'a F = ...

    fun Fmap f (VAR x) = VAR(x)
      | Fmap f (LAM(xs,M)) = LAM(xs, f M)
      ...

    fun inj (VAR x) = Var(x)
      | inj (LAM(xs,M)) = Lam(xs,M)
      ...

    fun prj (Var x) = VAR(x)
      | prj (Lam(xs,M)) = LAM(xs,M)
      ...
  end
```

Here is a module that implements the `ANF` interface by representing ANF terms as abstract lambda terms

```
functor ANF(structure Lam : LAM) : ANF =
  struct
    structure Lam = Lam
    type var  = Lam.var
    type prim = Lam.prim
    type t = Lam.t
    datatype 'a F = ...
    val var  = Lam.inj o Lam.VAR
```

```
    val vars = List.map var
    val app  = Lam.inj o Lam.APP
    val lam  = Lam.inj o Lam.LAM
    val prim = Lam.inj o Lam.PRIM
    fun bnd (x,N,M) = app(lam([x],M),[N])

    fun inj (RET x) = (var x)
      | inj (FUN (x,(ys,N),M)) = bnd(x,lam(ys,N),M)
      ...

    fun prj x =
      case Prjs.prj2 x of
        Lam.VAR x => RET x
      | Lam.APP(Lam.LAM([x],M),[Lam.LAM(ys,N)])  =>
          FUN(x,(ys,N),M)
      ...
      | _ => raise (Fail "bug")
  end
```

Here is a module that implements the `CPS` interface by representing CPS terms as ANF terms

```
functor CPS(structure ANF : ANF) : CPS =
  struct
    structure ANF = ANF
    type var  = ANF.var
    type prim = ANF.prim
    type t = ANF.t
    datatype 'a F = ...
    fun inj' (HALT arg) = ANF.RET  arg
      | inj' (FUN  arg) = ANF.FUN  arg
      ...
    fun inj x = ANF.inj (inj' x)

    fun prj' (ANF.RET  arg) = HALT arg
      | prj' (ANF.FUN  arg) = FUN  arg
      ...
      | prj' _ = raise (Fail "bug")
    fun prj x = prj' (ANF.prj x)
  end
```

The `LamUtil` module provides implementations of pretty printers, equality, α-equality, and α-renaming as well as several other functions for general lambda terms.

```
functor LamUtil(structure Lam : LAM) =
  struct
```

```
    ...
    fun toString M = ...
    fun freeVars M = ...
    fun rename M = ...
    fun eq M M' = ...
    fun alpha_eq M M' = ...

  end
```

The module below implements the a-normalization algorithm presented by Sabry et al. (Flanagan *et al.*, 1993).

```
functor ANFNorm(structure Lam : LAM
                structure ANF : ANF) : sig
  val anf_normalize : Lam.t -> ANF.t
end = struct
  structure ANF = ANF
  val ret  = ANF.inj o ANF.RET
  val call = ANF.inj o ANF.CALL
  val jump = ANF.inj o ANF.JUMP
  val fun' = ANF.inj o ANF.FUN
  val prim = ANF.inj o ANF.PRIM
  open Lam

  datatype cont_arg =
    V of Var.t
  | A of (Var.t * Var.t list)

  fun init (V x) = ret x
    | init (A(x,ys)) = jump(x,ys)

  fun norm M k =
    (case prj M of
       (VAR x) => k (V x)
     | (LAM (xs,M)) =>
         let val f = Var.fresh "f"
             val M = norm M init
         in fun'(f,(xs,M),k (V f))
         end
     | (APP(M,Ms)) =>
         norm_name M (fn x =>
         norm_names Ms (fn ys =>
           k (A(x,ys)))))
     | (PRIM(p,Ms)) =>
         let val t = Var.fresh "t"
         in norm_names Ms (fn xs =>
```

```
              prim(t,(p,xs),k (V t)))
         end

  and norm_name M k =
    norm M (fn (V x) => k x
             | (A(x,y)) =>
               let val t = Var.fresh "t"
               in call(t,(x,y),k t)
               end)

  and norm_names [] k = k []
    | norm_names (M::Ms) k =
        norm_name M   (fn x =>
        norm_names Ms (fn xs =>
          k (x::xs)))

  fun anf_normalize M = norm M init
end
```

The module below implements a straightforward CPS conversion expressed as a higher-order fold over ANF terms.

```
functor CPSCvt(structure ANF : ANF
               structure CPS : CPS) : sig
  val cps_convert : ANF.t -> CPS.t
end = struct
  structure CPS = CPS
  val halt = CPS.inj o CPS.HALT
  val jump = CPS.inj o CPS.JUMP
  val fun' = CPS.inj o CPS.FUN
  val prim = CPS.inj o CPS.PRIM
  open ANF
  structure Rec = Rec(structure T = ANF)

  fun convert M k = let
    fun f (RET x) k = k x
      | f (FUN(x,(ys,N),M)) k = let
          val kv = Var.fresh "k"
          val ys = kv::ys
          fun ret y = jump(kv,[y])
        in fun' (x,(ys,N ret),M k)
        end
      | f (CALL(t,(x,ys),M)) k = let
          val ret = Var.fresh "ret"
          val N = jump(x,ret::ys)
        in fun'(ret,([t],M k),N)
```

```
        end
      | f (JUMP(x,ys)) k = jump(x,ys)
      | f (PRIM(t,(p,xs),M)) k =
          prim(t,(p,xs),M k)
  in Rec.fold f M k
  end

  fun cps_convert M = convert M halt
end
```

Finally we provide a module that implements the `COMPILER` interface. The module is parameterized by the implementation of general lambda terms. Inside the module it is apparent that both CPS and ANF terms are represented as general lambda terms. We take advantage of this fact and reuse as much code as possible. Externally because we use an opaque signature constraint all the general terms, ANF terms, and CPS terms will be seen as distinct abstract types to clients of the resulting module.

```
functor Compile(structure Lam : LAM) :> COMPILE =
  struct
    structure Lam = Lam
    structure ANF = ANF(structure Lam = Lam)
    structure CPS = CPS(structure ANF = ANF)
    structure LU  = LamUtil(structure Lam = Lam)
    structure AN  = ANFNorm(structure Lam = Lam
                            structure ANF = ANF)
    structure Cvt = CPSCvt(structure ANF = ANF
                           structure CPS = CPS)

    val lam_toString  = LU.toString
    val lam_freeVars  = LU.freeVars
    val lam_eq        = LU.eq
    val lam_alpha_eq  = LU.alpha_eq
    val lam_rename    = LU.rename

    val anf_normalize = AN.anf_normalize
    val anf_toString  = lam_toString
    val anf_toLam     = (fn x => x)
    val anf_rename    = lam_rename

    val cps_convert   = Cvt.cps_convert
    val cps_toString  = anf_toString
    val cps_toLam     = (fn x => x)
    val cps_toANF     = (fn x => x)
    val cps_rename    = anf_rename
  end
```