Staging Algebraic Datatypes

Tim Sheard and Iavor S. Diatchki

OGI School of Engineering
Oregon Health & Science University
20000 N.W. Walker Road
Beaverton, OR 97006-8921
sheard@cse.ogi.edu, diatchki@cse.ogi.edu

Abstract. We describe a mechanism for staging algebraic datatype definitions. This is a generalization of the notion of partially static data, and allows run-time code generators to generate more efficient code. The idea is to capture static information, made evident by dynamic case analysis, and propagate this information into the arms of the case. This information is captured as a partially static description of the case expression's scrutinee, and is used to generate improved code. We give techniques for staging datatypes in the staged programming language MetaML, and several examples of its use. We also discuss semantic framework in which partially staged data can be analyzed.

1 Introduction

A high-level way to construct program generators is to use a staged programming language like MetaML[13, 9, 10]. MetaML adds staging annotations to standard ML. The staging annotations partition the terms of the program into stages, and are used by the programmer to change the order of evaluation of a program's terms. This can make the program more efficient, since it allows the programmer to remove redundant or inaccessible computation. This should come as no surprise to those who have studied the λ- calculus, since the number of steps in a reduction is strongly influenced by the order of evaluation. Staging annotations are thus seen as a convenient mechanism for influencing the control flow of an algorithm in a dimension not accessible through other techniques.

Our experience with MetaML has shown us that while it is good to stage the *control* of an algorithm, it is sometimes even better to stage its *data* as well. In this paper we discuss how datatype definitions can be staged, and how this can be used to build better program generators. We also discuss a formalism in which describes a semantic description of staged datatypes.

2 Program generators as staged programs

Program generators (meta-programs) are often used to address a whole class of related problems. Each problem has a separate, but related solution. The generator constructs another program (an object-program) that solves a particular instance. Usually the generated (object) program is "specialized" for the particular problem instance and uses less resources than a general purpose, non-generated solution. Program generators can be easily written using a staged programming language like MetaML. MetaML adds staging annotations to ML's standard features. These annotations supply an abstract user interface for constructing, manipulating, and executing code as a first class object. We discuss three of them here.

The staging annotations partition the program into stages. Brackets (`< _ >`) surrounding an expression lift the surrounded expression to the next stage (building code). Escape (`~( _ )`) (which should only appear within brackets) drops its surrounded expression to a previous stage (splicing in already constructed code to build a larger piece of code). Lift (`lift _`) evaluates its argument to a ground value (a first order value like `5`) and constructs a program in the next stage with that constant value. In a two stage world, bracketed code is dynamic, and unbracketed code is static.

Program generators remove a certain level of interpretive overhead, and can significantly improve run-time performance of a solution. The staging removes unnecessary or redundant computation that would occur in a generic solution. For example, without staging, to compute `ans` the algorithm unfolds the `pow` function many thousands of times.

```
fun pow n x =
  if n=0 then 1
         else x * (pow (n-1) x)

fun iota n m = if n '>' m then [] else n::(iota (n+1) m);

val ans = map (pow 3) (iota 1 10000);
```

But with staging, its is possible to unfold the function once, and only compute the cube many times, thus removing the overhead associated with the recursive calls to `pow`.

```
fun pow1 n x =
  if n=0 then <1> else < ~x * ~(pow1 (n-1) x) >;

val ans = map (run <fn x => ~(pow1 3 <x>)>) (iota 1 100);
```

This is possible because the term `<fn x =>  (pow1 3 <x>)>` evaluates at generation time to `<(fn a => a * a * a * 1)>`. The annotations direct the algorithm to perform some computations only once (or not at all), and hence drastically affect the control flow of the program.

Our experience with MetaML has led us to believe that annotating the control flow of an algorithm with staging annotations can be made *even more effective* by annotating the *data structures* as well. In this paper we discuss *how* this is done, and *why* is is useful to do so.

Consider the append function on lists, defined in MetaML as follows:

```
(* app: 'a list -> 'a list -> 'a list *)
fun app [] ys       = ys
  | app (x::xs) ys  = x :: app xs ys
```

This function is defined inductively on its first argument. We could exploit this fact to define a *staged* version of append:

```
(* app1: 'a list -> <'a list> -> <'a list> *)
fun app1 [] ys      = ys
  | app1 (x::xs) ys = <x :: ~(app1 xs ys)>
```

Note that app is identical to app1 except for the addition of the staging annotations bracket (< _ >) and escape (~ _). The function app1 is a runtime code generator, generating code when given its two arguments. For example

```
metaml> app1 [1,2] <[4,5]>;
<[1,2,4,5]>

metaml> <fn z => ~(app1 [3,6] <z>)>;
<fn z => 3 :: 6 :: z>
```

Intuitively, app1 works as follows: since the first argument of append is known at generation time, (for example [1,2]) the recursive part of the algorithm can be performed to produce a generated piece of code. In the case of the second example, we use app1 to generate a function. When this function is later applied, at run- time, it performs the appending without any of the overhead associated with the recursion.

This is similar to what partial evaluators, such as CMIX [1], do. The programmer identifies which of the inputs are static, and which are dynamic. The partial evaluator does some analysis to determine what results may be computed statically and which parts of the computation need to be performed at run time. In MetaML, the programmer does the analysis using his brain, and manually inserts the staging annotations < > and ~.

There is something unsatisfactory about the definition of app1. The above definition does not capture that the final result is not completely dynamic — the entire front part of the result is known (or at least *was* known — once it is embedded into code it cannot be recovered). It looks like what we need is a *partially static list*. With this in mind we could define another version of lists and yet another version of append which exploits this new version of lists:

```
datatype 'a list1 = Nil | Cons of ('a * 'a list1) | Later of <'a list>

(* app2: 'a list -> 'a list1 -> 'a list1 *)
fun app2 [] ys      = ys
  | app2 (x::xs) ys = Cons (x, app2 xs ys)
```

Note how app2 involves the use of both the old style of list, and the new style list1. What we have done is to *stage the data* instead of the function! The definition of app2 is identical to the definition of the original app, except for the different constructors of the staged data type. The type list1 contains all static and dynamic lists, but it also has other partially static lists. There are many other types, which are "between" the completely static 'a list and the completely dynamic <'a list>, and list1 captures this distinction.

We call algebraic data structures containing dynamic components *partially static datatypes*. Partially static *datatypes* are different from partially static *data*. Partially static data refers to parameterized datatypes where the parameter is code. Here, the *structure* is statically known, but (some of) the elements stored in it are dynamic. For example <int> list is partially static data. In partially static datatypes some (or all) of the structure is unknown. An example of such a datatype is int list1. In the rest of this paper we will examine different ways of staging data, and give examples to illustrate why this technique can be used to generate better code.

3 Staging datatypes

In this section we examine the process of staging a datatype. One way to proceed is to add more constructors to the datatype's definition. Here is for example one way to stage the sum type of ML.

```
datatype ('a,'b) Either
    = InLeft of 'a
    | InRight of 'b
    | InLeftC of <'a>
    | InRightC of <'b>
    | SumC of <('a,'b) sum>
```

The additional constructors allow the construction of partially static instances of Either, and allow the expression of additional information at *code generation time*. The first two constructors correspond to the usual Left and Right of the sum type. They cover the case when *all* information is known statically — both the *shape* and the *contents* are available. However, it is quite possible that we only know the shape of the value, but not the actual content. To represent such cases, we use the InLeftC and InRightC constructors. Here is an example of how that information might be used:

```
(* f: <('a,'b) sum> -> (('a,'b) Either -> <c>) -> <c> *)
fun f x k = <case ~x of
               Left c  => ~(k (InLeftC <c>))
             | Right c => ~(k (InRightC <c>))>
```

The function f performs a *run-time* check, and then passes the partially known result to the continuation k. Now k can produce better code, as it knows the shape of the value at code generation time.

Finally, it is possible that *nothing* is known about a value. We make such values with the SumC constructor. It contains the *code* which will produce the value.

Products can be staged in a similar way:

```
datatype ('a,'b) Pair
    = Pair of ('a *'b)
    | PairL of (<'a> *'b)     | PairR of ('a * <'b>)
    | PairLR of (<'a> * <'b>) | PairC of <'a *'b>
```

The constructor `Pair` makes fully static pairs. The constructors `PairL`, `PairR`, and `PairLR`, make pairs in which the left, right, or both components, respectively, are dynamic. At first it might seem a little strange that we have both `PairLR` and `PairC` — one might think that because we always know the shape of a pair, only having `PairLR` is enough. To illustrate the difference between the two consider the following example:

```
fun loop() = loop()

fun f (PairLR (x,y))    = x
  | f (PairC c)         = < #1 ~c>

metaml> run (f (PairLR (<1>,<loop()>)))
1
metaml> run (f (PairC <(1,loop())>))
...loops forever...
```

The difference between the two occurs when we have a `PairLR` value. Then, there are two pieces of code, which may each be run separately to generate the components of the pair. For example, if we do not need to compute the second component, we never need to run the corresponding piece of code. On the other hand, when we have a `PairC` value, there is only a single piece of code, which when run produces both components of the pair at once.

Next we stage lists. This is the first example where we have a recursive datatype. Following the pattern established in the above examples, here is our first attempt:

```
datatype 'a List
    = Nil
    | Cons of ('a * 'a List)
    | ConsL of (<'a> * 'a List)
    | ConsR of ('a * <'a List>)
    | ConsLR of (<'a> * <'a List>)
    | ConsC of <'a * 'a List>
    | Cons1 of <'a * 'a List>
    | ConsL1 of < <'a> * 'a List >
    | ConsR1 of <'a * <'a List> >
    | ConsLR1 of < <'a> * <'a List> >
    | ConsC1 of < <'a * 'a List> >
```

We associate `Nil` with the role played by `Left` and `Cons` with the role played by `Right`. Clearly there is something unsatisfactory about this. The size of the new staged datatype grows quadratically with the number of constructors and their sub-components. For small datatypes, or for ones where each constructor has only zero or 1 sub component this might work, but even for lists, with only two constructors it begins to get somewhat unwieldy. The datatype contains repetitions (e.g. `ConsC` and `Cons1`), and some of the constructors would only be useful, if we were interested in working with more than two stages of computation.

Another approach to staging a datatype is to follow the pattern we used with `list1` above. For each datatype construct another *shadow* version which co-exists with the original datatype. It has one additional constructor. For example, by using a convention of naming the shadow version of the type, and its constructors, with a suffix "`S`" (for staged) and naming the additional constructor the same as the type, we obtain:

```
datatype 'a List = Nil  | Cons of 'a * 'a List;
datatype 'a ListS = NilS | ConsS of 'a * 'a ListS | ListS of <'a List>;
```

Here the additional constructor has a single component which is code of the original (un-shadowed) type. This is a rich type which captures most, but not all stagings of lists. In particular it does not capture where the structure is known to be a `Cons` but the argument is a dynamic pair. This scheme is sufficient for most purposes. In Section 7 we give a complete staged type, and a formal algorithm for deriving it.

For types that contain components of primitive types (like `int` or `string`) we can use another strategy. Consider:

```
datatype 'a Partial = Stat of 'a | Dyn of <'a> * 'a list;
```

We can use `Partial` to describe different stagings of primitive types. We discuss the `('a list)` component of `Dyn` in Section 2. Then when an algebraic datatype embeds a primitive type, its staged version uses the `Partial` type constructor.

```
datatype Term =
   Int of int
 | Con of string * Term * Term

datatype TermS =
   IntS of (int Partial)
 | ConS of (string Partial) * TermS * TermS
 | TermS of <Term>;
```

Here `TermS` is a partial version of `Term` – every `Term` has a representation as a `TermS`, and every partially static term is also a member of the set described by the type `TermS`.

4 An application for staging – Term rewriting

One may think of a *term-rewriting system* as a set of directed rules. Each rule is made of a left-hand side and a right-hand side. Both the left-hand side and right-hand side of a rule are made of patterns. A pattern is a term with pattern matching variables as sub terms.

A rule $(a \Rightarrow b)$ may be applied to a term t if a sub term s of t matches a under some substitution σ. A rule is applied by replacing s with $\sigma\ b$ inside t to produce t'. We say *"t rewrites (in one step) to t'"*, and write $t \Rightarrow t'$. As an example, here are the rules for a Monoid [3]:

$$\begin{array}{lrcl} r_1: & x+0 & \Rightarrow & x \\ r_2: & 0+x & \Rightarrow & x \\ r_3: & x+(y+z) & \Rightarrow & (x+y)+z \end{array}$$

Variables x, y, and z in the rules can each match any term. If a variable occurs more than once on the left-hand side of a rule (a non-linear pattern), all occurrences must match identical terms. This occurs when a pattern has more than one occurence of a variable, such as in $(x+x)$.

Generally, the rules do not change over the life of the system. At the same time, the basic form of the matching function is a simultaneous traversal of a subject term and the left-hand side pattern of the rule it is being matched against. This offers an opportunity for staging: We can "specialize" matching over the rules in a first stage, and eliminate the overhead of traversing the pattern. Not only that, but as we will see, we can also remove a significant amount of administrative computations involved in constructing and applying the substitution σ. One would expect that this would significantly speed up the rewriting system.

5 Pattern matching

In this section we discuss the pattern matching sub-component of the general term-rewriting problem. First, we give an unstaged pattern matching function. This introduces the data structures and techniques we will use. Second, we stage this function. This produces a code generator, that when given a pattern generates a matching function (a piece of code). Third, we introduce an extension to the problem – the introduction of compound patterns constructed by *and-ing* and *or-ing* primitive patterns. Fourth we show that this is easily accommodated by our solution, but the generated results are not very efficient. Finally we introduce a partially static datatype to replace the datatype representing terms, and show how this leads to much more efficient generated matching functions.

In Figure 1 is an unstaged MetaML implementation of pattern matching over terms. Terms are represented by the datatype `Term`, and patterns by the datatype `Pat`. The type `Pat` has a structure similar to `Term`. We use the convention of deriving the names of the `Pat` constructor functions by suffixing the `Term` constructor functions with the letter "P". Note that `Pat` has one additional constructor function `VarP`, used to represent pattern variables.

A substitution is represented by a list of (`string` × `Term`) pairs. Pattern matching either succeeds with a substitution, or fails. This is encoded in the datatype `Result`. We have parameterized both `Subst` and `Result` over the type

```
datatype Term = Int of int  | Con  of (string * Term * Term)
datatype Pat  = IntP of int | ConP of (string * Pat  * Pat ) | VarP of string;
type 'a Subst       = (string * 'a) list
datatype 'a Result  = Fail | Ok of 'a Subst

(* Term -> Term -> bool *)
fun termEq (Int x) (Int y) = x = y
  | termEq (Con (con1,x1,y1)) (Con (con2,x2,y2))
        = (con1 = con2) andalso (termEq x1 x2) andalso (termEq y1 y2)

fun match1 (VarP x) t env k =
      (case lookup x env of
         NONE    => k (Ok ((x,t)::env))
       | SOME t' => if termEq t t' then k (Ok env) else k Fail)
  | match1 (IntP x) (Int y) env k =
      if x = y then k (Ok env) else k Fail
  | match1 (ConP (conp,xp,yp)) (Con (con,x,y)) env k =
      if conp = con
         then match1 xp x env (fn Fail    => k Fail
                               | Ok env' => match1 yp y env' k)
         else k Fail
  | match1 _ _ env k = k Fail
```

Fig. 1. Unstaged pattern Matching

of terms. This will be useful in our later versions, as we will replace the `Term` datatype with a staged version.

The function `match1` (`1`, because it is the first of several versions) takes a pattern to match a against, a subject term, a substitution, and a continuation. The continuation is expecting a `Term Result`. It proceeds by a simultaneous traversal over the structure of both the pattern and the subject term. If the structure of the two match, an `Ok` result is built and the continuation is applied. If the structures don't match, the continuation is applied to `Fail`.

One additional point is worth making. In the `VarP` case, if the variable already appears in the substitution then we need to test that the term it binds, matches the current subject term. This is the job of the term equality function `termEq`. This occurs if the pattern is non-linear, i.e. having the same pattern variable appearing more than once, as in $(x + x)$.

The `match1` function can be employed by applying it to a pattern, a term, an empty environment, and the identity continuation:

```
fun test p t = match1 p t [] id
```

To illustrate the power of staging, we rewrite `match1` by adding staging annotations: bracket, escape, and lift.

```
fun match2 (VarP x) t env k =
      (case lookup x env of
         NONE    => k (Ok ((x,t)::env))
```

```
      | SOME t2 => <if termEq ~t ~t2 then ~(k (Ok env))  else ~(k Fail)>)
  | match2 (IntP x) term env k =
      <case ~term of
         Int i => if i = ~(lift x) then ~(k (Ok env)) else ~(k Fail)
       | Con (con,x,y) => ~(k Fail) >
  | match2 (ConP (conp,xp,yp)) term env k =
       <case ~term of
          Int i => ~(k Fail)
        | Con (con,x,y) => if con = ~(lift conp)
            then ~(match2 xp <x> env (fn Fail    => k Fail
                                      | Ok env' => match2 yp <y> env' k))
            else ~(k Fail)>
  | match2 _ _ env k = k Fail
```

This version of matching, `match2`, takes an *unknown* term `t` as input. Since `t` is a piece of code representing a computation that will produce a `Term` at run-time (as opposed to generation-time), we cannot inspect the structure of the term. The best we can do is splice `t` in to the generated code. We splice it into contexts that will, at run-time, do the inspection. Note how the recursive calls to `match2` are escaped, and use sub patterns, and the bound variables of the generated code as arguments.

By applying `match2` to a pattern, an empty substitution (note that substitutions now bind variables to `<Term>`), and an appropriate continuation we generate a matching function where all interpretive overhead of walking over the pattern is eliminated.

```
metaml> p1;
val it = ConP ("+",VarP "x",VarP "x") : Pat;

metaml> val matchfun = test2 p1;
val matchfun =
<(fn Int d => NONE
   | Con(c,b,a) =>
        if c = "+"
           then if termEq a b then SOME ([("x",b)]) else NONE
           else NONE)>
 : <Term -> (string * Term) list option>
```

Running `test2` on larger patterns, generates larger more complex matching functions, but the basic idea remains the same.

5.1 An extension

We can extend the usefulness of pattern matching by allowing the creation of complex patterns, built with pattern operators applied to primitive patterns. To illustrate the point we add two new constructors to the `Pat` datatype.

```
datatype Pat = VarP of string | IntP of int | ConP of (string * Pat * Pat)
             | AndP of (Pat * Pat) | OrP of Pat * Pat
```

We extend the match2 function by adding two new cases, one for each of the new constructors of Pat. We show only those two cases here.

```
fun match2 (VarP x) (t: <Term> ) env k =
  . . .
    | match2 (AndP(p1,p2)) term env k =
        match2 p1 term env (fn Fail   => k Fail
                             | Ok e2  => match2 p2 term e2 k)
    | match2 (OrP(p1,p2)) term env k =
        match2 p1 term env (fn Fail   => match2 p2 term env k
                           | Ok e2  => k (Ok e2))
  . . .
```

For an AndP pattern, continue to match the second pattern only if the first pattern succeeds. Pass the substitution computed by the first pattern onto the matching of the second pattern. For an OrP pattern, try the second pattern only if the match on the first fails. The ease in which this was accomplished, explains in part, why we choose a continuation based implementation of matching. Unfortunately, the code generated is not as elegant as the generator.

```
metaml> p3;
val it = OrP (ConP ("-",VarP "x",VarP "y"),ConP ("+",VarP "x",VarP "y")):Pat
metaml> -| test2 p3;
val it =
<(fn a =>
     (case a of   (* CASE 1 *)
       Int i =>
          (case a of   (* CASE 2 *)
            Int m => NONE
          | Con(l,k,j) => if l = "+" then SOME ([("y",j),("x",k)])
                                     else NONE)
     | Con(d,c,b) =>
          if d = "-"
             then SOME ([("y",b),("x",c)])
             else (case a of   (* CASE 3 *)
                    Int h => NONE
                  | Con(g,f,e) => if g = "+" then SOME ([("y",e),("x",f)])
                                             else NONE)))>
 : <Term -> (string * Term) list option>
```

Note that the variable a is the scrutinee of three separate case expressions, each of which test the same thing. Cases 2 and 3, are completely redundant, since they are both inside the arms of Case 1, and the structure of a is known in those contexts. Indeed, Case 2 is not only redundant, its Con arm is dead code, since the context implies that a is an Int term. The same can be said of the Int arm of Case 3.

These redundant and dead code-fragments are all generated by a continuation inside a generated case. Unfortunately the generator does not know about this contextual information. The key observation here is the following, the term t is dynamic (i.e. a piece of code), and nothing is known about its structure. But once the generator moves into a context under one of the arms of a case, *part of* `t`*'s structure becomes known* within that context. This is why we can re-represent `t` as a partially static datatype.

The strategy is to define a partially static version of `Term`, and to extend the structure of a `Result`, so that results contain partially static information about the term "matched so far". This information is built from information implied by the context evident in arms of dynamic case expressions. To enable this we define the datatype `TermS` which is a partially static version of `Term`, and redefine `Result` so that each constructor has a `TermS` as a sub component.

```
datatype TermS =
    IntS of int Partial
  | ConS of (string Partial) * TermS * TermS
  | TermS of <Term>;

datatype Result  = Fail of TermS | Ok of (TermS Subst * TermS);
```

The strategy then calls for rewriting the staged match function with a partially static `TermS` as the subject term, rather than just a completely dynamic `<Term>`, and a continuation with the redefined `Result` type as its domain. Thus every continuation has the information to discover what shape the local context tells about the subject term.

5.2 The match3 function

The new version can be found in Figure 2. The function `match3` has many new cases corresponding to different partially static subject terms. For each pattern constructor (`IntP`, and `ConP`), there are several cases for various forms of the partially static constructors (`IntS`, `ConS`, and `TermS`). The cases with the `TermS` constructors are completely analagous to the case found in the `match2` function, since the `TermS` constructor has only a completely dynamic sub component. Let's consider the fourth case of `match3` as typical. Below, we contrast it with the corresponding case of the `match2` definition.

```
  match2 (IntP x) term env k =
      <case ~term of
         Int i => if i = ~(lift x) then ~(k (Ok env)) else ~(k Fail)
       | Con (con,x,y) => ~(k Fail) >


  match3 (IntP x) (TermS term) env k =
      <case ~term of
         Int i => if i = ~(lift x)
                    then ~(k (Ok(env,IntS (Stat x))))
                    else ~(k (Fail (IntS (Dyn (<i>,[x])))))
       | Con (con,x,y) =>
           ~(k (Fail(ConS (Dyn (<con>,[]),TermS <x>,TermS <y>)))))>
```

```
fun matchArgs conp xp yp x y env k = match3 xp x env
      (fn (Fail t) => k (Fail(ConS (Stat conp,t,y)))
        | (Ok(env1,t)) => match3 yp y env1
             (fn (Fail t1) => k (Fail(ConS (Stat conp,t,t1)))
               | (Ok(env2,t')) => k (Ok(env2,ConS (Stat conp,t,t')))))
and match3 (VarP x) t env k =
      (case lookup x env of
         NONE    => k (Ok ((x,t)::env, t))
       | SOME t2 => <if ~(eqTermS t t2) then ~(k (Ok(env,t)))
                                        else ~(k (Fail t))>)
  | match3 (IntP x) (term as (IntS (Stat y))) env k =
      if x = y then k (Ok(env, term)) else k (Fail term)
  | match3 (IntP x) (term as (IntS (Dyn (y,isNot)))) env k =
      if elem x isNot
         then k (Fail  term)
         else <if ~y = ~(lift x) then ~(k (Ok(env,IntS (Stat x))))
                                 else ~(k (Fail(IntS (Dyn (y,x::isNot)))))>
  | match3 (IntP x) (TermS term) env k =
      <case ~term of
         Int i => if i = ~(lift x) then ~(k (Ok(env,IntS (Stat x))))
                                   else ~(k (Fail (IntS (Dyn (<i>,[x])))))
       | Con (con,x,y) => ~(k (Fail(ConS (Dyn (<con>,[]),TermS <x>,TermS <y>)))))>
  | match3 (ConP (conp,xp,yp)) (term as (ConS (Stat con,x,y))) env k =
      if conp = con
         then matchArgs conp xp yp x y env k
         else k (Fail term)
  | match3 (ConP (conp,xp,yp)) (term as (ConS (Dyn (con,isNot),x,y))) env k =
      if elem conp isNot
         then k (Fail term)
         else <if ~con = ~(lift conp)
                  then ~(matchArgs conp xp yp x y env k)
                  else ~(k (Fail(ConS (Dyn (con,conp::isNot),x,y)))) >
  | match3 (ConP (conp,xp,yp)) (TermS term) env k =
      <case ~term of
         Int i => ~(k (Fail(IntS (Dyn (<i>,[])))))
       | Con (con,x,y) => if con = ~(lift conp)
          then ~(matchArgs conp xp yp (TermS <x>) (TermS <y>) env k)
          else ~(k (Fail(ConS (Dyn (<con>,[conp]),TermS <x>,TermS <y>)))))>
  | match3 (AndP(p1,p2)) term env k =
      match3 p1 term env (fn (Fail t) => k (Fail t)
                           | (Ok(e2,t)) => match3 p2 t e2 k)
  | match3 (OrP(p1,p2)) term env k =
      match3 p1 term env (fn (Fail t) => match3 p2 t env k
                           | (Ok(e2,t)) => k (Ok(e2,t)))
  | match3 pat term env k = k (Fail term)
```

Fig. 2. Matching with partially static subject term

The primary difference is in the structure of the `Result` passed to the continuation `k`. In `match2` no information about the subject term is passed to the continuation. In `match3` a partialliy static `TermS` is embedded in both the `Fail` and `Ok` results. Note how the partially static term describes exactly what the context of the dynamic test reveals. For example if the subject term dynamically matches the structure required by an `IntP` pattern, and if the value of the integer matches the patterns integer, then the partially static `TermS` encodes this: `(IntS (Stat x))`. On the other hand, if the integer subcomponents don't match, then the encoding is: `(IntS (Dyn (<i>,[x])))`.

Recall the defintion of the `Partial` datatype:

```
datatype 'a Partial = Stat of 'a | Dyn of <'a> * 'a list
```

The `Dyn` constructor indicates an unknown dynamic value. This is appropriate when the test of equality on the integers `i` and `x` fail. However, there is some information known about `i` – we know that it is not equal to `x`! We use the second sub component of the `Dyn` constructor to record this.

The other cases make use of the partially static information painstakingly propagated in the `Result`s. They use the static information to improve the code generated. As a typical example of one of these, consider the fourth case of `match3`.

```
match3 (ConP (conp,xp,yp)) (term as (ConS (Dyn (con,isNot),x,y))) env k =
    if elem conp isNot
        then k (Fail term)
        else <if ~con = ~(lift conp)
            then ~(matchArgs conp xp yp x y env k)
            else ~(k (Fail(ConS (Dyn (con,conp::isNot),x,y)))) >
```

Here the subject term is known to be a `Con` term. Its operator name is unknown, but we do know a list of operator names to which it is not equal. Thus there is no need to generate a `case` expression to test if it might possibly be an `Int` term. We also know that if the pattern operator name is in the list `isNot`, then the test must fail. Otherwise we test it against pattern operator, and either succeed, or fail again, but this time with even more information about what it is not.

The number of cases more than doubles, but this cost does not come without benefit. The resulting generated matching functions are signficantly smaller, and more efficient. Compare the result generated by `match2` on the test pattern `p3` (on page 10), with:

```
metaml> test3 p3;
val it =
<(fn Int d => NONE
   | Con(c,b,a) =>
      if c = "-"
         then SOME ([("y",a),("x",b)])
         else if c = "+" then SOME ([("y",a),("x",b)]) else NONE)>
 : <Term -> (string * Term) list option>
```

The redundant tests, and all the dead code has been eliminated.

6 Related work

The use of partially static data as the input descriptions for partial evaluation systems originates in the work of Mogenson[8]. Mogenson used context-free grammar descriptions to describe such partial structures. The importance of such descriptions was understood even earlier, but Mogensen's was the first analysis that could recover such information.

Consel [2] used an analysis based upon cons-points, i.e. labels labelling every constructor function in a program, to build similar graph-like descriptions, where the nodes of the graph are the cons-points.

Launchbury[7] used projection analysis to assign each partially static object in a typed language, to a point in a lattice of partial domains. John Hughes[4] work on type-based specialization may also be closely related.

At some point, the term *partially static* took on the association of static data in the parameter position of parametric, types such as tuples, and lists. But, the idea of arbitrary partialness was apparent in all three of these early works. The authors are unaware of any work where such partial types are exploited to build better generated programs.

Recently, one of the authors[11] of this paper used a code type constructor, to describe similar information, as the input to an automatic binding-time analysis.

The work in this paper can be seen as an attempt to make such information available to, and controllable by the programmer. In a manually staged language, such as MetaML, it is crucial that the programmer can control all aspects of the process. It is hard to see how such information could be discovered in any automatic way.

Many program generators use a post-generation, optimization stage, where rules are used to optimize code after it is generated. This is problematic in a system with run-time code generation, since if a language allows both an eval-mechanism, and intensional analysis of generated code (the ability to observe the structure of code after it is generated) it loses confluence, and hence cannot be deterministic[12].

The techniques illustrated in this paper show how this optimization phase can be incorporated into the generation phase. In essence performing the optimization in parallel with the code generation.

7 Discussion and future work

In this paper we have used the word partial to mean "not fully". Thus a partially static datatype is one where not all of the structure is static, and some sub structure remains dynamic. The word partial can also be used to describe non-terminating or error producing computations (i.e. not fully total). This is interesting because there are many simlilarities between the two. Both lead to a lattice of types. In the static case the bottom element is the code type, signifying no information about the value in the current stage. In the computation case the bottom element signifies no result, ever. We believe this analogy to be a useful one, and have begun to investigate its possibilities.

An important property of a staged version of a datatype, is that it should capture every possible partial description of the original type. That is every fully static value, the completely dynamic version, and every point in between. While this is desireable from a semantic point of view, it is not necessary from a practical point of view.

Their should be a total injection function from the original type to the staged version, but only the partially static points of the lattice, which are useful to the computation, need be captured. The recipe given in Section 3 meets this criteria for all the examples we have used. In the next section we describe an attempt to formalize a algorithm that given a datatype defintion constructs another datatype definition which is a partially static version of the orginal.

7.1 Datatype staging mechanism

A staged version of a datatype is a new datatype with one additional constructor. Except for this additional constructor, the domain type of each staged constructor is replaced by the staged type of the domain of the original constructor. The additional constructor contains code, which when executed, will produce a value of the original type. Since both staged and original types coexist in the staged version of the datatype, we need to maintain this relationship. We do this with the aid of a constraint system. We shall use Haskell (with extensions), as we can use its type classes mechanism [5] to keep track of the constraints.

We first define the two-parameter class `Staged`. It is a relation between types — two types α and β are related by `Staged`, if β is the staged version of α. We use a functional dependency [6] to indicate that, `Staged` is in fact a type function — if we know the base type, we also know the staged type.

```
class Staged a b | a -> b where
    inj :: a -> b
```

Since Haskell does not have the staging features of MetaML, we are going to represent code as just the identity type constructor.

```
newtype Code a = Code a
```

For basic types with many constructors (e.g. integers and characters) we use the trick from the previous section:

```
data Partial a = Stat a | Dyn (Code a) [a]
type IntS = Partial Int
type CharS = PartialChar
```

Now we can define some instances, which will enable us to discharge the `Staged` constraints for basic types:

```
instance Staged Int IntS where
    inj = Stat

instance Staged (Code Int) IntS where
    inj x = Dyn x []
```

The instances for `Char` are essentially the same. Next we show how to stage a simple data-type — the booleans.

```
data Bool = True | False
data BoolS = TrueS | FalseS | BoolS (Code Bool)

instance Staged Bool BoolS where
    inj True = TrueS
    inj False = FalseS

instance Staged (Code Bool) BoolS where
    inj = BoolS
```

It is slightly more complicated to stage a parameterized data-type. We illustrate how to do this using Haskell's `Either` type.

```
data Either a b = Left a | Right b

data EitherS a b
    = forall a'. Staged a a' => LeftS a'
    | forall b'. Staged b b' => RightS b'
    | EitherS (Code (Either a b))
```

We have used existential types here (although they are written as `forall` in Haskell!). For every parameter to the datatype we postulate that there is a staged version of it. What is stored in the constructors is this staged type. The type of `LeftS` for example is:

```
LeftS :: Staged a a' => a' -> EitherS a b
```

Now we can define the instances for discharging the `Staged` constraints for the `Either` type.

```
instance (Staged a a', Staged b b')
    => Staged (Either a b) (EitherS a b) where
    inj (Left x) = LeftS (inj x)
    inj (Right x) = RightS (inj x)

instance Staged (Code (Either a b)) (EitherS a b) where
    inj = EitherS
```

Finally we look at how to stage a recursive type. It turns out this is not much more difficult: just as before, we inductively stage the subcomponents of each constructor; type parameters are handled with existential types as above; finally recursive calls to the origianl type are replaced by recursive calls to the staged type.

```
data List a = Nil | Cons a (List a)

data ListS a
    = NilS
    | forall b. Staged a a' => ConsS a' (ListS a)
    | ListS (Code (List a))

instance (Staged a a') => Staged (List a) (ListS a) where
    inj Nil = NilS
    inj (Cons x xs) = ConsS (inj x) (inj xs)
```

We leave the incororation of these ideas into an actual staged language, such as MetaML, as future work. Until that time users can hand-stage their own datatypes capturing whatever partially static information their particular application demands.

8 Conclusion

We have demonstrated a technique for staging algebraic datatypes. A staged version of an algebraic datatype captures all possible partial descriptions of the original type. The partially static datatype makes it possible to capture static information, made evident by dynamic case analysis, and propagate this information into the arms of the case.

In some sense this is the dual of lifting, making dynamic information static. While every static value can be completely lifted to a dynamic value, the information made possible by dynamic tests often only reveals the top level of structure. A mechanism to capture this mixture of static and dynamic information is necessary. Partially static datatypes provide this structure. By making more information static, it is possible to generate smaller, better, and more efficient residual programs.

We gave techniques for deriving staged versions of datatypes in the staged programming language MetaML, and illustrated its use in the staging of complex patterns.

9 Acknowledgments

The work described here grew out of a collaboration between Tim Sheard and Mitch Wand on staging an Aspect Oriented Programming Language semantics. Tim would like to thank Mitch for lively discussions about the uses of staging and pattern matching, and his help in refining the idea on Mitch's visit to OGI in January of 2001.

The work was supported by NSF Grant CCR-0098126, the M.J. Murdock Charitable Trust, and the Department of Defense.

References

1. L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994. (DIKU report 94/19).
2. C. Consel. Binding time analysis for higher order untyped functional languages. In *1990 ACM Conference on Lisp and Functional Programming*, pages 264–272. ACM, ACM Press, June 1990.
3. N. Dershowitz. Computing with rewrite systems. *Information and Control*, 65:122–157, 1985.
4. J. Hughes. Type specialisation for the λ-calculus; or, A new paradigm for partial evaluation based on type inference. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation. Dagstuhl Castle, Germany, February 1996*, volume 1110 of *Lecture Notes in Computer Science*, pages 183–215. Berlin: Springer-Verlag, 1996.
5. M. P. Jones. *Qualified Types: Theory and Practice*. PhD thesis, Oxford University, July 1992. Also available as Programming Research Group technical report 106.
6. M. P. Jones. Type classes with functional dependencies. *Lecture Notes in Computer Science*, 1782:230–??, 2000.
7. J. Launchbury. *Projection Factorizations in Partial Evaluation*. PhD thesis, University of Glasgow, Nov. 1989.
8. T. A. Mogensen. *Partially static structures in a self-applicable partial evaluator*. North-Holland, 1988.
9. T. Sheard. *Advanced Functional Programming*, chapter Using MetaML: A staged Programming Language, pages 207–240. Springer, 1999.
10. T. Sheard. Accomplishments and research challenges in meta-programming. In W. Taha, editor, *Proceedings of the Workshop on Semantics, Applications and Implementation of Program Generation (SAIG'01)*, volume 2196 of *LNCS*, pages 2–44, Berlin, September 2001. Springer Verlag. Invited talk.
11. T. Sheard and N. Linger. Search-based binding time analysis using type-directed pruning. Technical report, OGI School of Engineering, Oregon Health & Science University, 20000 NW Walker Road, Beaverton OR 97006-8921, Feb. 2002.
12. W. Taha. *Multi-Stage Programming: Its Theory and Applications*. PhD thesis, Oregon Graduate Institute of Science and Technology, Nov. 1999.
13. W. Taha and T. Sheard. Multi-stage programming with explicit annotations. In *Proceedings of the ACM-SIGPLAN Symposium on Partial Evaluation and semantic based program manipulations PEPM'97, Amsterdam*, pages 203–217. ACM, 1997.