Sequentializing Programs Defined by Pattern Matching (DRAFT)

Todd Simpson[1]
Research and Development
AGT Limited
Calgary, AB, CAN
T2G 4Y5
simpson@cpsc.ucalgary.ca

Robin Cockett[2]
Department of Computer Science
University of Calgary
Calgary, AB, CAN
T2N 1N4
robin@cpsc.ucalgary.ca

July 21, 1992

Abstract

A new approach to translating programs defined using pattern matching into sequential form is presented. The technique is based on the theory of decision tree reduction.

A program definition using patterns is a parallel specification and does not determine a sequential evaluation. A problem arises, therefore, in attempting to preserve as far as possible the intended parallel meaning during the translation to sequential form. In particular, a sequentialized program may be forced to evaluate patterns which need not be evaluated when parallelism is available. It is known that some definitions using patterns, called *sequentializable* definitions, can be translated without semantic loss, and algorithms to sequentialize these definitions have been documented. In this paper we present an algorithm which works uniformly over *all* definitions—it handles sequentializable definitions correctly, but also optimizes the sequential realization of other definitions.

1 Introduction

Pattern matching provides a convenient syntax for programming languages. Unfortunately, it also complicates the semantics and, in the case of functional languages, can compromise equational reasoning and laziness. These complications arise as programs defined by pattern matching have a parallel semantics which is being realized on a sequential machine.

Traditionally definitions by pattern matching may include overlapping patterns, constant patterns, nested patterns, and multiple arguments. Sometimes conditional or guarded equations, and non-linear patterns are also allowed, although we will not discuss these aspects in this paper. Further, pattern lists which are not exhaustive can easily be completed, so we assume all our pattern lists are complete. Thus, a typical pattern matched definition for our purposes is:

[1]Partially supported by the Natural Sciences and Engineering Research Council of Canada, under an Industrial Research Fellowship.

[2]Supported by the Natural Sciences and Engineering Research Council of Canada.

$$\begin{array}{llll} h & True & True & = 1 \\ h & True & \_ & = 2 \\ h & False & \_ & = 3 \end{array}$$

h is the name of the program being defined; each row is called an equation; $True\ True$ is an example of a pattern; and 1 is an example of a right-hand-side (rhs). As we are concerned only with linear patterns (that is, no repeated variables are allowed), variable names not appearing in the rhs are unimportant and are indicated by the anonymous variable _.

An intuitive operational meaning for a call $h\ x\ y$ is as follows: Imagine a set of machines attempting to match each pattern against $x\ y$. One machine is assigned to each constructor in the pattern and evaluates the appropriate argument to see if it matches. If all the constructors mach, then the rhs is selected. If any constructor does not match, then a match failure is indicated, even if other machines working on other constructors in the same pattern have not yet returned a result. If there are overlapping patterns, (i.e., $h\ True\ True$ can match either the first or second equation), then a means of choosing just one equation must be given. Traditionally, the first equation (in the syntactic ordering of the definition) to match is selected. This implies that all previous equations (in the same ordering) must terminate with a failure to match before a rhs can be selected.

This intuition is intended to reflect a parallel scheme which does the least amount of evaluation necessary to choose a rhs, thus avoiding nontermination whenever possible. However, this is not the only possible scheme, and indeed, does not capture the laziest possible semantics.

Early pattern matching translation algorithms did not attempt to achieve laziness, even for sequentializable definitions[3]. The first such (documented) algorithm is that of Augustsson [1], and a similar one is derived by Wadler [11]. More recently algorithms which are lazy for sequentializable definitions have been presented by Laville [7], Puel and Suarez [8], and Sekar [9].

In this paper we introduce another approach to the problem using the theory of decision tree reduction developed by Cockett [4]. This approach is lazy for sequentializable definitions, but also optimizes the laziness of non-sequentializable ones—something which previously has been handled in an ad-hoc manner. Further, in defining the class of sequentializable functions previous work has considered only the patterns in the equations, not the rhs's (following the lead set by Huet and Levy [6]). We show that when rhs's are considered, the class of sequentializable functions is not as large and reflects a more realistic view of parallel evaluation. These ideas are discussed in the next section.

2 Parallel Pattern Matching

Let us formalize the operational meaning discussed above. We will assume an underlying type system including at least:

$$\alpha\ ::=\ 1 \mid \alpha \rightarrow \alpha \mid (\alpha \times \ldots \times \alpha)$$

Patterns are then defined as:

$$\begin{array}{lll} pat & ::= & () :: 1 \ \mid\ \mathsf{variable} :: \alpha \ \mid\ \_ :: \alpha \\ & \mid & (pat :: \alpha_1, \ldots, pat :: \alpha_n) :: (\alpha_1 \times \ldots \times \alpha_n) \\ & \mid & (\mathsf{constructor} :: (\alpha \rightarrow \alpha')\ pat :: \alpha) :: \alpha' \end{array}$$

Expressions representing rhs's will be written as $e :: \alpha$ with appropriate sub/superscripts. Likewise, $t :: \alpha$ will represent an arbitrary expression to match over, and $c :: \alpha \rightarrow \alpha'$ a constructor. When

[3]Also called *strongly sequential definitions*

dealing only with patterns and rhs's the name of the program being defined is unimportant and we will write (often omitting the type information)

$$\{(p_1 :: \alpha, e_1 :: \alpha'), ...(p_n :: \alpha, e_n :: \alpha')\}$$

as an ordered set (the syntactic ordering) of equations.

We define M to indicate if a given pattern matches a term. The result of M is one of three things: F which is a failure to match; T^i_σ, which is a successful match where σ is the substitution to apply to the i^{th} rhs; or nontermination ($\perp$).

$$\begin{array}{lcl}
M_i(v,t) & = & T^i_{[v:=t]} \\
M_i(\_,t) & = & T^i_{[]} \\
M_i((),()) & = & T^i_{[]} \\
M_i((p_1,...,p_n),t) & = & M_i(p_1,\pi_1(t)) \otimes ... \otimes M_i(p_n,\pi_n(t)) \\
M_i(c(p),d(t)) & = & \left\{\begin{array}{ll} M_i(p,t), & c = d \\ F, & otherwise \end{array}\right. \\
M_i(c(p),v) & = & M_i(c(p_1,...,p_n), eval(v))
\end{array}$$

Here π_j is the j^{th} projection of a tuple and $eval$ evaluates a term just enough to find its top level constructor—subterms are not evaluated (weak head normal form). $\otimes$ is a parallel-and defined as

$$x \otimes y = \left\{\begin{array}{ll} \perp, & (x = \perp) \wedge (y = \perp) \\ F, & x = F \\ F, & y = F \\ T^i_{\sigma++\sigma'}, & (x = T^i_\sigma) \wedge (y = T^i_\sigma) \end{array}\right.$$

where ++ is list append. The use of parallel-and in M ensures that a pattern can fail to match even if some nonterminating components of t are evaluated. Using M, $Meval$ can be defined as:

$$Meval\{(p_1,e_1)...(p_n,e_n)\}(t) = \oplus_{i=1,..,n}\langle[\neg_i(\otimes_{q_j \in Q_{e_i}} \neg_j(M_j(q,t)))] \otimes M_i(p_i,t)\rangle$$

$$Q_{e_k} = \{p_j \mid j < k\}$$

where $\oplus$ is parallel-or and $\neg_i$ is defined as:

$$\begin{array}{lcl}
\neg_i(T^j_\sigma) & = & F \\
\neg_i F & = & T^i_{[]}
\end{array}$$

This definition warrants some explanation. Below is an annotated rhs for the case that p_i is the matching pattern:

$$\oplus_{i=1,..,n}\langle \quad \overbrace{[\neg_i(\otimes_{q_j \in Q_{e_i}} \neg_j(M_j(q,t)))]}^{\text{all patterns in } Q_{e_i} \text{ terminate with } F} \quad \otimes \quad \overbrace{M_i(p_i,t)}^{p_i \text{ matches}} \ \rangle$$

Q_{e_i}, as defined above, includes all patterns before p_i in the syntactic ordering. Thus, all previous matches must terminate with F before the match on p_i will be accepted. This means that the top-to-bottom ordering is implemented and that at most one clause in the outer $\oplus$ can be true. If a member of Q_{e_i} does not terminate, neither will the match on p_i.

It follows that the less elements which Q_{e_i} contains, the better the chances for a terminating match. One way to reduce Q_{e_i} is to remove all patterns which to not overlap with p_i. This will reduce the size of Q_{e_1} but will not affect termination as, if the match on p_i terminates with T^i_σ, then the matches on any overlapping pattern would have terminated with F. Another way to reduce the size of Q_{e_i}, which does affect termination, is to consider rhs's. Assume two equations have structurally equivalent rhs's, where some standard naming procedure has been employeed to ensure that variables bound in the patterns are handled correctly (this is described in section 3.1.1). If a match against the first pattern fails to terminate, then $Meval$, as defined above, will also fail to terminate. But, as the second equation has the same rhs, we could also attempt a match against it, which if successful will return the correct result regardless of the fact that a previous match failed to terminate. We can encode this change in the algorithm by redefining Q_{e_k} to

$$Q_{e_k} = \{p_j \mid j < k, \quad e_j \neq e_k\}$$

Thus, Q_{e_k} is a smaller set implying that fewer previous matches have to terminate. We illustrate the effect on a simple example:

$$\begin{array}{llll} and & True & True & = & True \\ and & False & \_ & = & False \\ and & \_ & False & = & False \end{array}$$

Using the first strategy $and \perp False$ will not terminate as the first argument must be evaluated in the second equation. However, with the latter strategy this will terminate as the reliance on the second equation terminating has been removed.

The more rhs's which are equivalent, the smaller Q_{e_k} becomes. Thus, a more defined equality test is lazier than a less defined one. If semantic equality is undecidable, so therefore is the laziest solution.

The rest of the paper is devoted to designing a translation into sequential form which captures as much of the laziness in the second form of $Meval$ as possible.

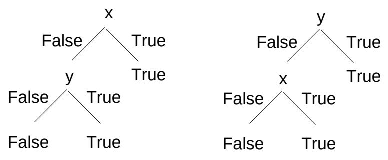
3 Sequential Forms

We would like to realize this parallel operational semantics for pattern matched definitions on a sequential machine. Unfortunately, it is not always possible to preserve these semantics. Consider the definition

$$\begin{array}{llll} or & False & False & = & False \\ or & \_ & \_ & = & True \end{array}$$

To sequentialize this one must decide which argument of or to touch first. If the chosen argument fails to terminate, so will the sequential form. With the parallel semantics, if the other argument terminated with $True$, a value would have been returned.

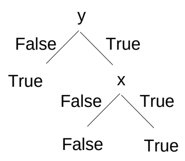
The sequential forms of a pattern matched definition may be represented by decision trees. For $or\ x\ y$, two such trees are possible



In these trees, "laziness" is indicated by arguments in the pattern which do not occur on a path from root to leaf (called a *spine*). When nested patterns are present, embedded patterns will occur in the appropriate branches under the top level constructor.

Sometimes it is the case that a sequential form exists which preserves the parallel semantics.

$$\begin{array}{llllll} f & True & True & = & A \\ f & \_ & False & = & B \\ f & \_ & True & = & C \end{array}$$



Here, for a call $f\ x\ y$, y must always be evaluated, so it will be the root of the decision tree. Only if y is $True$ must we evaluate x.

Definitions by pattern matching which can be sequentialized have been studied in [6, 7, 8], and can be recognized easily. However, many definitions can not be sequentialized and need not be of the *and/or* form—consider the example given by Berry [2]:

$$\begin{array}{llllll} Berry & A & A & \_ & = & X \\ Berry & B & \_ & A & = & Y \\ Berry & \_ & B & B & = & Z \\ Berry & \_ & \_ & \_ & = & W \end{array}$$

There are many sequential forms for this example, based on the order of evaluation of the arguments. While none of these forms respect the parallel semantics, some forms are lazier than others. The approach we present finds a lazier form—the result for Berry's example is given in section 6.

There is one other aspect of these definitions which is worth mentioning—it is also possible to optimize definitions based on the parallel semantics. That is, there may exist a parallel form which expresses more laziness more succinctly than the original form. For example, the definition of *and* given earlier can be optimized to

$$\begin{array}{lllll} and & True & True & = & True \\ and & \_ & \_ & = & False \end{array}$$

which is obviously more efficient. We are also interested in this aspect of the problem [10], although it is not presented here.

Rather than trying to generate a good sequential form immediately, the next section introduces a straightforward sequentialization. Then, in section 4 this form will be optimized into a lazier form.

3.1 Generating a Sequential Form

In this section we look at an algorithm for generating decision trees. The nodes of the tree are (indices of) expressions/subexpressions to evaluate. The leaves are the rhs's, and the branches are labeled by constructors. While this form is easily translated into if-then-else's or simple case statements, the decision tree form lends itself to optimization [4], which is discussed in the next section. The result of generating a tree as discussed in this section does not necessarily produce an optimal form. It is the subsequent optimization of the tree that reintroduces some laziness.

Generating a decision tree from a set of patterns is the most straightforward form of sequentialization. Wadler [11] presents the most well known translation of this sort. We present a different algorithm in order to simplify its correctness statement and to give optimization a head start.

While the trees generated from matching definitions may be exponentially large in the worst case, optimization of the tree is $O(n\ log\ n)$ in the size of the tree n.

For this section we will work with a particularly simple source language—we are only interested in case statements (which will contain our pattern matching), and variables. We begin by discussing what the generated decision trees must look like.

3.1.1 Dependencies and Variable Bindings

To build the tree we must decide what the types at the nodes and leaves will be. The nodes must encapsulate the dependencies in patterns, for example, a cons cell (:) must be evaluated before the first element of that cons cell can be looked at. The leaves must contain not only the expression to evaluate, but also binding information for variables bound in the matching process.

As patterns can be viewed as trees, nodes will be indices into trees indicating constructors which must be evaluated. An example is shown below:

$$(x \overbrace{\;:\;}^{0} (\overbrace{T}^{010} \overbrace{\;:\;}^{01} \overbrace{nil}^{011}))$$

Each element of a tuple is assigned a unique index. The dependency relation on nodes is the standard prefix ordering on indices.

Deciding on the type for the leaves is less straightforward. The driving motivation is to ensure that (structural) equality can be checked at the leaves. A naive approach, ignoring variable bindings, would cause too many leaves to be identified:

```
case   x       of
       (y:x)   =>   x
       _       =>   x
```

In this example, although both rhs expressions are structurally equal, the x is bound differently in each case. It is also not sufficient to record the index at which a variable occurs, as shown below:

```
case   z            of
       (con1 x y)   =>   x
       (con2 x y)   =>   x
```

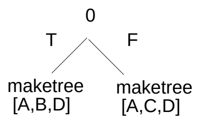
Here, in both cases the index of x in the patterns is 00, but they are bound under different constructors. The simple solution is to ensure that variables are bound to the same node in the tree; this means that as well as having equal indices, the labels above both are also equal. Thus, at the leaves of the tree we maintain the expression to evaluate and a full description of the node to which each variable is bound. This information can be calculated during the the generation of the tree—thus, the tree itself need not contain binding information; all variables can be considered _'s. Nevertheless, for exposition we retain variable names.

3.1.2 *Maketree*; generating a decision tree

The generation of a decision tree can now be explained by way of an example. We will call the algorithm *maketree*.

```
case   (x,   y)           of
       (v,   (w:nil))     =>   A v w
       (T,   (w:x:y))     =>   B w x y
       (F,   (w:x))       =>   C w x
       (v,   w)           =>   D v w
```

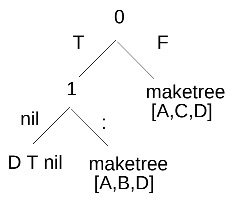
We will reference a given equation by the capitalized function on the rhs (for example the first equation is A). The indices are $x = 0$ and $y = 1$. We start with x, note that it is of type *boolean* and thus will have two branches labeled T and F. We form two new groups of equations to process in each branch, namely all those which match T, $[A, B, D]$, and all those which match F, $[A, C, D]$. The order of the equations is inherited from the original definition so that first-fit is propagated. So far, the decision tree looks like (where *maketree* $[A, B, D]$ loosely refers to the compilation of the equations A, B and D).



The equations in each group are modified to reflect the knowledge of the value of x When x is T, the equations for A, B and D become:

$$\begin{array}{llll} case & y & of & \\ & (w : nil) & => & ATw \\ & (w : x : y) & => & Bwxy \\ & w & => & DTw \end{array}$$

The algorithm recurses on these new equations, the only new difficulty being nested patterns. In the case above, 1 (i.e., y) would become the next decision point with nil and : being branches.



For the case of :, A, B and D become:

$$\begin{array}{lllll} case & (y10, & y11) & of & \\ & (w, & nil) & => & ATw) \\ & (v, & w : u) & => & Bvwu \\ & (\_, & \_) & => & DTy) \end{array}$$

where $y10$ and $y11$ have been introduced simply to indicate that the two new indices being matched over are 10 and 11.

When complete, the resulting *case* statements will represent a decision tree. The algorithm is given in Figure 1 (where binding information is encoded in the rhs's as indices into the tree). It has obvious similarities and differences from the one found in [11]. In particular, our algorithm freely duplicates rhs's (although some of these will be removed through optimization) and a given index occurs at most once in a spine of the resulting tree.

We delay looking at the correctness of *maketree* until we have formally defined decision trees and some of their properties.

4 Decision Tree Reduction

It remains to optimize the decision tree generated by *maketree* in order to reintroduce as much laziness as possible. Fortunately, there is a well defined theory for manipulating decision trees, developed in part by the second author [4], which meets our needs. We quickly review this theory here—for more indepth reading see [4]. We start by formalizing the definition of decision trees:

```
data ExpTy       = Ecas [String] [([PatTy String String],ExpTy)]
                 |...
data PatTy a b   = Con a [PatTy a b]
                 | Var b                          deriving Eq
data Dtree a b c = Node a [(b,Dtree a b c)]
                 | Leaf c                       deriving Eq

     -- (vb ...) precalculates binding information for variables
maketree (Ecas xs pel) = td (map (\x -> [x]) (from02 xs)) (vb (from02 xs) pel)
 where
  from02 xs = take (length xs) (from 0)

     -- (constructors pel) is the list of type constructors for index 0
  td []     (([],(vb,e)):l) = Leaf (vb,e)
  td (n:ns) pel             = case (constructors pel) of
                                [] -> td ns (map knock1 pel)
                                _  -> Node n subnodes
   where
    knock1 (((Var (v,vs)):pl),(vb,e)) = (pl,(vb++[(v,vs)],e))

    subnodes = map (\ (c,g) -> (c,td ((new c)++ns) g)) (grp pel (constructors pel))
     where new c = map (\x -> n++[x]) (take (arity c) from0)

           grp pel = map (\c -> (c,concat (map (\p -> chk p c) pel)))
            where chk ((Con c pl):pl',(vb,e)) d  | c == d    = [(pl++pl',(vb,e))]
                                                 | otherwise = []
                  chk ((Var (v,vs)):pl,(vb,e)) d = [((nv d)++pl,(vb++[(v,vs)],e))]

                  nv d = take (arity d) (repeat (Var ("_",[])))
```

Figure 1: Translation from case statement to decision tree (in Haskell)

Definition 1 *A* **decision tree** *is a term of a "decision theory", which is an algebraic theory with operation symbols interpreted as* **cooperations***:*

$$q :: A \to A_1 + \cdots + A_n$$
$$q(a) = a_i, \qquad 1 \le i \le n$$

A decision theory can be defined in order to reason about decision trees. Trees are interpreted as terms and the theory introduces equalities over terms.

Definition 2 *A* **decision theory***, $D = (Q, E)$, consists of of a set, Q, of operation symbols (or decisions) and a set, E, of equations. Each $q \in Q$ has an arity, $arity(q) \ge 2$. The* **terms** *of D are defined recursively by:*

1. *Each $x \in V$ is a term, where V is a set of variables, and*
2. *Given terms, $t_1, ..., t_n$, and $q \in Q, arity(q) = n$, then $q[t_1 \mid ... \mid t_n]$, sometimes abbreviated $q[t_{1..n}]$ or $q[t^{1..n}]$, is a term.*

Each equation $(t_1 = t_2) \in E$ indicates an identity of terms. E must contain at least the following three identities for all operation symbols q, q_1, q_2:

Idempotence:

$$q[x \mid ... \mid x] = x$$

Repetition:

$$q[x_{1..r-1} \mid q[y_{1..m}] \mid x_{r+1..m}] = q[x_{1..r-1} \mid y_r \mid x_{r+1..m}]$$

Transposition:

$$q_1[q_2[x^1_{1..m}] \mid ... \mid q_2[x^n_{1..m}]] = q_2[q_1[x_1^{1..n}] \mid ... \mid q_1[x_m^{1..n}]]$$

A little study will show these three to be very reasonable. Idempotence indicates that if a decision has no effect on the outcome, then the decision is not needed. Repetition notes that once a decision has been made, if that decision occurs again, the outcome is already known. Finally, transposition indicates that if two decisions must be made, we can switch their order in the tree.

Given a decision tree t, a **spine** for t is a path from the root to a leaf, including the leaf itself. The set of all spines for t is written $SP(t)$. A decision tree, t, is **repeat reduced** if, for any $s \in SP(t)$, no decision q_i occurs more than once in s. A tree can be repeat reduced using the three basic identities [3], and is written as $rpt(t)$. A decision tree, t, is **idempotent reduced** if the idempotence relation has been fully applied in the simplifying direction. After application of this reduction a tree is written $idp(t)$. A tree, t, is termed **simply reduced** if it is both repeat and idempotent reduced, $idp(rpt(t))$ (not $rpt(idp(t))$ as this may not be idempotent reduced).

While idempotent reduction and repeat reduction can "reduce" the size of the tree, transposition has no affect on the size. However, a transposition may change the tree so one of the other operations can be applied. For example, $q_1[q_2[x \mid y] \mid q_2[x \mid z]]$ is simply reduced, but transposition gives $q_2[q_1[x \mid x] \mid q_1[y \mid z]]$ which may be idempotent reduced to $q_2[x \mid q_1[y \mid z]]$.

Thus, the three identities lead to a preordering on trees.

4.1 Irreducible Trees

An **irreducible** tree is a "best" tree in that it is a least tree for any given reasonable preorder.

Definition 3 *Define a* **reasonable preorder**, *$\leq_r$, on trees as the least preorder that is:*

Monotonic: Whenever $t_1 \leq_r t_2$,

$$q[... \mid t_1 \mid ...] \leq_r q[... \mid t_2 \mid ...]$$

Idempotent reducing:

$$t \leq_r q[t \mid ... \mid t]$$

Repetition reducing:

$$q[x_{1..r-1} \mid y_r \mid x_{r+1..m}] \leq_r q[x_{1..r-1} \mid q[y_{1..m}] \mid x_{r+1..m}]$$

Transposition invariant:

$$q_1[q_2[x^1_{1..m}] \mid ... \mid q_2[x^n_{1..m}]] \leq_r q_2[q_1[x_1^{1..n}] \mid ... \mid q_1[x_m^{1..n}]]$$

Intuitively, a tree is irreducible if there is no means of rewriting the tree using the identities in E such that the tree is smaller under $\leq_r$. The significance of $\leq_r$ is discussed in [4]. For our purposes we note that eliminating nodes is a reasonable preorder, and doing so will reduce the amount of evaluation encoded in a tree. Thus, an irreducible tree represents a sequentialization which is *as least as lazy* as the tree generated by *maketree*. We will see below that the trees can in fact be much lazier.

An algorithm to reduce a tree to an irreducible form is described in [4]. The essence of the algorithm is to use transposition to manipulate the tree into a form where idempotence can be applied. This it does by trying to push decisions down towards the leaves until an idempotent reduction is found.

More formally, a decision q is **semi-essential** in a tree t if and only if there exists a t' with q as a root decision such that $t =_d t'$. A term is **semi-essential simple** if the only semi-essential decision is the root. A term is *factored* by decision q if the subterm at every occurrence q is semi-essential simple. In [4] is is shown that a tree t is **irreducible** if and only if t factored by q cannot be idempotent reduced, for every decision q in t.

Given a repeat reduced tree, t, the reduction algorithm in Figure 2 reduces t to an irreducible form. The algorithm is $O(n \ log \ n)$ in the size of the tree, n. To provide guidance we note that `branch_reduce` performs repeat reduction for a single decision. `pull_up` places a node at the top of the tree, hangs the entire tree off each branch, and then uses `branch_reduce` to eliminate further occurrences of the new root (effectively carrying out transposition). `semi_essentials` finds nodes which can be placed at the root without changing the decision structure (i.e. those on which transposition can be applied). `elimination` performs idempotent reduction, and carries out the placement and reduction of new roots. `reduce` applies this procedure recursively through the tree, starting at the leaves.

4.2 Application to Pattern Matching

The trees we create with *maketree* from patterns are slightly more complicated than those discussed above.

```
data Dtree a b = Node a [Dtree a b]
               | Leaf b              deriving (Eq)

d_fold lf nd (Leaf n)   = lf n
d_fold lf nd (Node n t) = nd n (map (d_fold lf nd) t)

branch_reduce n m = d_fold Leaf (\n' r -> if n == n' then r!!m else Node n' r)

pull_up (n,Node n' bl)) =
    Node n (fst (foldl (\(r,m) _ -> (r++[branch_reduce n m (Node n' bl)],m+1)) ([],0) bl))

semi_essentials = d_fold (\x -> []) (\n r -> n:(foldr1 intersect r))

elimination (Leaf n)                                = (Leaf n)
elimination (Node n ((Leaf x):l)) | all (== (Leaf x)) l = (Leaf x)
elimination (Node n l)                              = nd (semi_essentials (Node n l))
      where nd (_:a:l') = Node n' r
                             where (Node n' l'') = pull_up (a, Node n l)
                                   r             = map elimination l''
            nd    _     = Node n l

reduce = d_fold Leaf (\n r -> elimination (Node n r))
```

Figure 2: Decision Tree Reduction (in Haskell)

- There are dependencies between nodes in the decision trees generated from function definitions. For example, an expression must be evaluated to a *cons* cell before the *head* of that cell can be evaluated.
- The branches in our decision trees will have to be labeled (by constructors) as opposed to ordered.

The second point is fairly trivial; instead of using the ordering of branches to do repeat reduction we must look at the labels on the branches. The first point is more fundamental and requires that the dependencies are introduced into the theory. A simple way to do this, which assumes that variable renaming has been done to avoid conflicts, is given below. The full details will be given in [10] following the ideas described in [5].

Definition 4 *An* **ordered decision theory**, $D' = ((Q, \ll), E)$ *where* $\ll$ *is a partial order on* Q*, has terms defined by*

1. *Each* $x \in V$ *is a term, where* V *is a set of variables, and*
2. *Given terms* $t_1, ..., t_n$ *and* $q \in Q$ *where there do not exist any* q' *in* t_i *where* $q \ll q'$ *and* $arity(q) = n$*, then* $q[t_{1..n}]$ *is a term.*

The equations in the theory include idempotence and repetition as before, but a restriction arises on transposition, namely:

Transposition:

$$q_1[q_2[x^1_{1..m}] \mid ... \mid q_2[x^n_{1..m}]] = q_2[q_1[x_1^{1..n}] \mid ... \mid q_1[x_m^{1..n}]] \quad if \quad q_2 \not\ll q_1$$

The same restriction applies when defining the reasonable preorder.

The ordering, of course, makes a change in the reduction algorithm, but it is surprisingly small. if $q' \ll q$, then q' can not be semi-essential in a tree containing node q. This is reflected in the code for generating semi-essentials:

```
semi_essentials = d_fold (\x -> [])
                         (\n r -> n:(filter (\a -> a << n) (foldr1 intersect r)))
```

5 Properties of Sequentialization

We have introduced both *maketree* and *reduce*. In this section we show that *maketree* encodes the original intent of the patterns and that sequentializable definitions are correctly translated. Further, we include a short section which gives an alternate characterization of irreducibles which illuminates the link with laziness.

5.1 Correctness of *maketree*

Our correctness statement involves termination. Informally, if a match on the sequential form terminates, the result should be equal to the result from matching on the parallel form. This can be formalized using *Meval* if we can recover patterns from the sequential form.

A spine for a labeled tree is a path from the root to a leaf including the labels and the leaf. For example,

$$0 \rightarrow^{c_1} 0.1 \rightarrow^{c_4} 1 \rightarrow^{c_2} rhs$$

is a spine where the c_i are constructor names.

Give a tree T and a spine $s \in SP(T)$, we can recover a pattern, say p_s, which corresponds exactly to that spine. In particular, each branch in the spine, say $i \rightarrow^c i'$, imposes the restriction that $p_s/i = c$. If j is not a node in the spine, then p_s/i must be a variable. The pattern corresponding to the spine above is thus $(c_1(\_, c_4(\_)), c_2(\_, \_, \_))$ (which relies on the arity of the constructors).

It is therefore possible to generate a list of equations from a tree. Let $pattern(s)$ be the pattern derived from the spine s and let $rhs(s)$ be the leaf associated with the spine. Define $patrhs(T) = \{(pattern(s), rhs(s)) \mid s \in SP(T)\}$.

We can now state a theorem for the correctness of *maketree*—namely, if the patterns recovered from the sequential form give a result, then the same result would have been found using the parallel semantics.

Write $E \leq_\perp E'$ if E' terminates more often than E, and define $Choose\ P\ t = e_i$ just when $Meval\ P\ t = T^i_\sigma$. Further, we write P^i_c for the set $\{(lift\ p, e) \mid (p, e) \in P, p/i = c \quad \vee \quad p/i = \_\}$, where

$$\begin{array}{lcl} lift(c(p_{11}, ..., p_{1n}), p_2, ..., p_m) & = & (p_{11}, ..., p_{1n}, p_2, ..., p_m) \\ lift(\_, p_2, ..., p_n) & = & (\underbrace{\_, ..., \_}_{11,...,1n}, p_2, ..., p_n) \end{array}$$

Theorem 1 *Let $P = \{(p_1, e_1), ..., (p_n, e_n)\}$. Then,*

$$Choose\ (patrhs(maketree\ P))\ t \quad \leq_\perp \quad Choose\ P\ t$$

Proof: By induction over the size (number of constructors) in $p_1, ..., p_n$.

Base: If $n = 0$ then $p_1 = v$ and

$$\begin{aligned} & Choose\ (patrhs(maketree\ \{(v, e_1)\}))\ t \\ = \ & e_1 \\ = \ & Choose\ \{(v, e_1)\}\ t \end{aligned}$$

Induction: Assume the theorem holds for pattern lists with less constructors. Assume that $t/0 = c_i$. Then,

$$\begin{aligned} & Choose\ (patrhs(maketree\ P))\ t \\ = \quad & Choose\ (patrhs(0[\rightarrow^c_1 (maketree\ P^0_{c_1})\ |\ ...\ |\ \rightarrow^{c_n} (maketree\ P^0_{c_n})]))t \\ = \quad & Choose\ (patrhs(0 \rightarrow^{c_i} (maketree\ P^0_{c_i})))t \\ \leq_\perp \quad & Choose\ P^0_{c_i}\ t \\ = \quad & Choose\ P\ t \end{aligned}$$

The final step follows as patterns not in $P^0_{c_i}$ can not match. $\square$

5.2 Sequentializable Definitions

Here we give a definition of sequentiality in terms of decision trees, and relate it to the existing notion over patterns. To do so, we must assume that the rhs's of the original equations were all distinct. We have seen how considering rhs's can affect sequentiality.

5.2.1 Simple Trees

A decision is **essential** in t if it is semi-essential in every t' which is decision equivalent to t. An occurrence of a decision is an **essential occurrence** in t if it is essential in the subtree at that occurrence. A tree is **simple** if every occurrence of every decision is essential.

In this section we will show that sequentializable definitions always reduce to simple trees. It is thus important to know if a tree is simple. A decision q is **delayed** in t if for each $s \in SP(t)$ either q is not in s, or it is the last decision in s. In [4] it is shown that: if q is delayed in a simply reduced t, then q is essential in t iff it is semi-essential in t.

The reduction algorithm easily spots semi-essentials, and delaying q is simply a matter of pulling up everything below q. This is $O(n)$ where n is the size of the tree. Thus, it is straightforward to decide if a tree is simple.

5.2.2 Sequential Patterns

As stated earlier, pattern matching is traditionally considered over just the patterns, not rhs's. Sequentiality for patterns was originally defined by Huet and Levy although we derive our presentation from Laville [7].

Meval actually performs evaluation of terms to achieve a match. The more traditional approach is to fail if the term is not sufficiently evaluated. We will call such a scheme *Match* and it is easily derived by changing the rhs of the last defining equation for M (in section 2) to F. With *Match* we use Q_{e_k} as the set of all previous patterns—this is the case where rhs's are not considered, and thus $Match$ can be considered to work only on the patterns. In the following, let $P = \{(p_1, e_1), ..., (p_n, e_n)\}$

Given a term, t, i is an **essential index**[4] of a match for P if

- t/i is a variable, and
- $Match\ P\ t = F$, and
- σ is a substitution such that $Match\ P(\sigma(t))$ is true

together imply that $(\sigma(t))/i$ is no longer a variable.

P is **sequential on** t if $Match\ P\ t = F$ and there exists a σ such that $Match\ P\ (\sigma(t))$ is true, together imply that P has an essential index in t. P is **sequential** if it is sequential on every term. Thus, a set of patterns is sequential if there is, at each step in the matching process, a subexpression which *must* be evaluated for the matching to continue.

5.2.3 Relating Terms and Trees

We would like to establish an equivalence between simple trees and sequential patterns. The main conceptual difficulty is in realizing that essential indices and essential decisions are really the same notion. We assume that the tree is generated by *maketree* and that the original rhs's were structurally unique.

Theorem 2 *Given a set of equations, $P = \{(p_1, e_1), ...(p_n, e_n)\}$, where the e_i are distinct, then P is sequential if and only if the reduced tree generated by maketree P, is simple.*

Proof:

($\rightarrow$) By induction over the size (number of constructors) of P. In the base case $P = \{(v, e_1)\}$ and $Match$ is always true. In the inductive case, as P is sequential, there is an essential index, say i. Then i is an essential node in $reduce(maketree\ P)$ and may be transposed to the root. Further, an instantiation of t/i to c_j will generate an essential index, i', which will be an essential root the subtree covering patterns $P^i_{c_j}$ (by the inductive hypothesis). As each node is essential in the subtree below it, the resulting tree is simple.

($\leftarrow$) The root of the tree, say i is essential and is an essential index for the variable term x. If x is instantiated to $c_j(v_1, ..., v_n)$, then the node under i by way of branch c_j will be an essential index for this new term. When a leaf is reached, no essential index is needed as all more defined terms will automatically match. $\Box$

If any rhs's are equated, then a set of patterns may be sequential while its tree representation is not. This is because the tree generated by *maketree* need not be idempotent reduced, and reduction may eliminate a node.

5.3 Reduction and Laziness

While it is clear that for sequential definitions an optimal tree is being generated, we would like to expand slightly on how nonsequential trees are also optimized. An irreducible tree is the "best" tree for a given reasonable preorder. We characterize reasonable preorders by the existance of maps between the spine sets of trees. It is shown that using the reduction algorithm always simplifies the spine sets.

[4]More usually called an **index**—we believe this term is more descriptive.

Definition 5 *If B_1 and B_2 are bags, then $B_1 \leq_b B_2$ if, for each $b \in B_1$, the number of occurrences of b in B_1 is less or equal the number of occurrences in B_2.*

A surjective map $e : SP(t_1) \longrightarrow SP(t_2)$ is a bag cover, if when $e(s_1) = s_2$, the leaves in s_1 and s_2 are equal, and $s_1 \leq_b s_2$.

From [4] we have:

Theorem 3 *If $t_1 \geq_r t_2$, then there is a bag-cover map $e : SP(t_1) \longrightarrow SP(t_2)$. Further, when t_2 is repeat reduced, the converse is also true.*

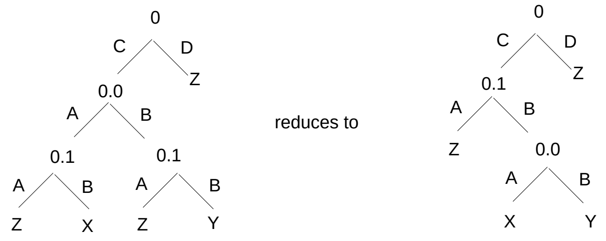
Thus, e encodes the fact that reduction reduces the number of steps to reach the leaves of the tree. For pattern matching, this also ensures that reducing a tree improves termination properties for both sequential and nonsequential functions (the bag cover property is extended in the obvious way to ordered and labeled trees).

6 Examples

We repeat some of the standard examples in our formalism.

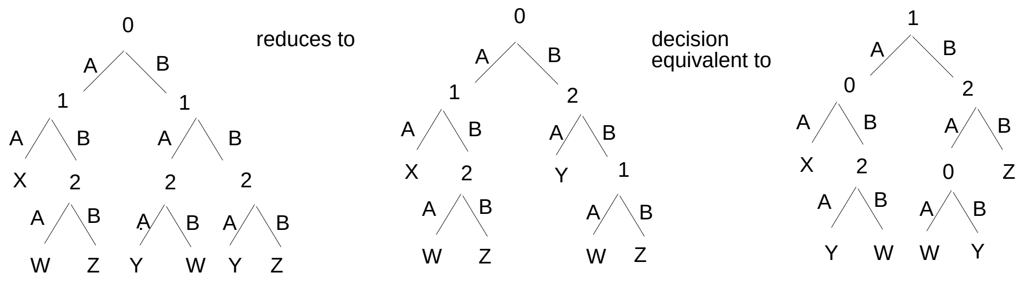
Example 1:

```
case    x         of
        C(A, B)   =>   X
        C(_, B)   =>   Y
        _         =>   Z
```



The first tree is generated by *maketree*, while the second is the simple tree found by reduction—i.e. the definition is sequentializable.

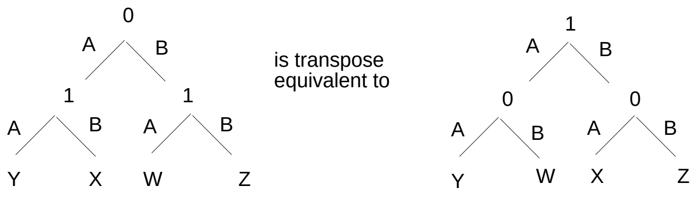
Example 2: Berry's example (defined earlier):



Other algorithms may fail to get an optimal representation (they may produce the first tree shown—as previously mentioned, these algorithms simply state that nonsequential functions can be added, not how this is done). The first tree reduces to an optimal form (the second tree), but is not simple as the third tree is decision equivalent but not a transpose.

Example 3: The second tree is decision equivalent to the one generated by *maketree*, but is also a transpose, and thus this example is sequential.

```
case  x  y  of
      A  B  =>  X
      A  _  =>  Y
      _  B  =>  Z
```



7 Conclusion

We have introduced a different approach to compiling and optimizing definitions by pattern matching. The two phase approach differentiates between the sequentialization strategy and the optimization of the resulting decision structure—this later phase being a straightforward application of decision tree reduction. The algorithm is fully lazy for sequentializable definitions and also improves the termination properties of nonsequentializable definitions. Thus, it improves upon the algorithms proposed by Laville [7] and Puel and Suarez [8]. Sekar et al. [9] had also noticed one particular application of transposition/idempotent reduction in relation to pattern matching—we have shown here how decision tree reduction extends and formalizes this.

While we have emphasized how reduction improves the laziness of a tree, efficiency and representation are also optimized by reduction. Therefore, this algorithm applies equally well within the framework of strict evaluation, or in systems with strong termination properties.

References

[1] L. Augustsson. Compiling pattern matching. In *Proc. 1985 Workshop on Implementations of Functional Languages, Goteborg, Sweden.* Chalmers University of Technology, February 1985.

[2] G. Berry. Stable models of typed lambdi-calculi. In G. Ausiello and C. Bohm, editors, *Proc. 5th Int. Conf. on Automata, Languages, and Programming*, volume 62 of *LNCS*. Springer Verlag, 1978.

[3] R. Cockett. Discrete decision theory: Manipulations. *Journal of Theoretical Computer Science*, 54:215–236, 1987.

[4] R. Cockett and J. Herrera. Decision tree reduction. *Journal of the ACM*, 37(4):815–842, October 1990.

[5] R. Cockett and T. Simpson. Distributive logic. In preparation.

[6] G. Huet and J. Levy. Call by need computations in nonambiguous linear term rewriting systems. Technical Report 359, INRIA, 1979.

[7] A. Laville. Lazy pattern matching in the ML language. volume 287 of *LNCS*, pages 400–419. Springer Verlag, 1988.

[8] L. Puel and A. Suárez. Compiling pattern matching by term decomposition. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, pages 273–281, 1990.

[9] R. Sekar, R. Ramesh, and I. Ramakrishnan. Adaptive pattern matching. Technical report, Preprint, SUNY Stoney Brook, 1992.

[10] T. Simpson and R. Cockett. On pattern matching. In preparation.

[11] P. Wadler. Efficient compilation of pattern matching. In S. Peyton-Jones, editor, *The Implementation of Functional Programming Languages*, chapter 5. Prentice Hall, London, 1986.