Pattern-driven Reduction in Haskell

William L. Harrison [1]

Pacific Software Research Center
OGI School of Science & Engineering
Oregon Health & Science University
Beaverton, Oregon, USA

Richard B. Kieburtz [2]

Pacific Software Research Center
OGI School of Science & Engineering
Oregon Health & Science University
Beaverton, Oregon, USA

Abstract

Haskell is a functional programming language with nominally non-strict semantics, implying that evaluation of a Haskell expression proceeds by demand-driven reduction. However, Haskell also provides pattern matching on arguments of functions, in **let** expressions and in the match clauses of **case** expressions. Pattern-matching requires data-driven reduction to the extent necessary to evaluate a pattern match or to bind variables introduced in a pattern. In this paper we provide both an abstract semantics and a logical characterization of pattern-matching in Haskell and the reduction order that it entails.

1 Introduction

Functional programming languages are often given a semantics in terms of a rewrite system inspired by Church's lambda calculus. Unlike the rewriting system of the lambda calculus, the rules for reduction of terms in a functional programming language do not generally have the property of confluence, but embody a strategy for evaluation order. Two popular such strategies are selection of a leftmost-innermost redex, known as "call-by-value", and leftmost-outermost, known as "call-by-name", or with the addition of a mechanism for reduction caching, as "lazy" evaluation.

[1] Email: `wlh@cse.ogi.edu`
[2] Email: `dick@cse.ogi.edu`

For pragmatic reasons, programming languages nearly always incorporate term structures whose evaluation by reduction require exceptions to the default strategy. These are typically terms that manifest some notion of control of evaluation, such as conditional or case expressions, or in a lazy language, operations on mutable state objects or I/O commands. This paper explains such a circumstance, arising in the pattern-driven evaluation of expressions in Haskell.

1.1 Pattern-matching in Haskell

Haskell is known as a lazy functional language, implying that the default reduction strategy used to evaluate terms to a head-normal form is leftmost-outermost. Furthermore, the data constructors of Haskell's algebraic data types are non-strict—a constructor application is a head-normal form even if the argument terms are unnormalized.

However, Haskell also employs pattern-matching to select among alternative branches of a case expression or alternative equations of a function definition. Pattern-matching can force partial evaluation of an argument to a function or a case discriminator (i.e., the "e" in "case e of . . ."), whether or not the value of any variable bound in the pattern is ever demanded. In that sense, the reduction order mandated by pattern-matching is not lazy and may deviate from a leftmost-outermost strategy.

Of course, situations arise in which a programmer "knows" that a certain pattern-match would succeed, so pattern-matching for control purposes (e.g., selecting a branch in a case expression) can be delayed until a variable within the pattern is demanded. By doing so, one makes the pattern-matching lazier and thus more in harmony with Haskell's default evaluation strategy. Haskell provides a prefix notation ($\sim$) with which a programmer can indicate that pattern-matching is disabled for control. A control-disabled pattern (called *irrefutable* in the Haskell literature) serves only to specify that variables occurring in the pattern are bound by a successful match. Such bindings can be evaluated lazily. Control-disabled patterns call for a reduction strategy different from nominal, pattern-driven reduction.

The reduction strategy employed to evaluate Haskell programs containing pattern constructs is implicit in the language definition, although a careful reading of the Haskell Report[5] is necessary to uncover this reduction strategy. It is perhaps the aspect of the Haskell language least understood by Haskell amateurs. This paper gives a semantics for the fragment of Haskell that involves pattern-matching, employing monads to encapsulate control effects. It also proposes inference rules of a programming logic for this language fragment. The logic is part of a larger framework, developing and implementing a property verification logic for the full Haskell language.

The semantics and the logic provide complementary but coherent formal descriptions that are consistent with the language definition [5]. They should help a reader to comprehend the nuances of Haskell reduction strategies.

```
type Name = String
data P  = Pvar Name | Pwildcard | Pcondata Name [P] | Ptilde P             {- Patterns -}
data E  = Var Name | Undefined | ConApp (Name,[LS]) [E] | Case E [(P,E)]     {- Expressions -}
data LS = Lazy | Strict deriving Eq
```

Fig. 1. Abstract Syntax of a Haskell Fragment

2 A Haskell fragment and its informal semantics

In this section, we give an overview of the Haskell fragment specified in this paper and give an informal description of its semantics. Although the fragment considered here is small, it is large enough to expose the relevant issues in Haskell's pattern-directed evaluation of expressions. The expressions we consider are case expressions and datatype constructor applications. Haskell datatypes may be declared with *strictness annotations*[5] and this perturbs the way patterns are matched and constructor applications are evaluated as we show below. Among the patterns we consider are the control-disabled patterns (i.e., patterns annotated with $\sim$). Figure 1 presents the abstract syntax of the fragment, written as two Haskell datatypes, `E` (for expressions) and `P` (for patterns).

2.1 Patterns in Haskell

Patterns may occur in several different syntactic contexts in Haskell—in case branches, explicit abstractions, or on the left-hand sides of definitions. We say that a pattern is *abstracted* if it occurs in an operand position on the left-hand side of a function definition[3], under a lambda-symbol (the backslash, in Haskell) or to the left of the arrow symbol (`->`) in a case branch. Since the roles played by abstracted patterns are similar in every context, we shall focus on patterns as they occur in case expressions.

The datatype `P` in Figure 1 defines the abstract syntax for the patterns we consider in this paper.

2.1.1 Variables and wildcard patterns

A variable is itself a pattern which matches any term[4]. Thus a match with a variable never fails and always accomplishes a binding. A term need not be evaluated to match with a pattern variable.

Haskell designates a so-called wildcard pattern by the underscore character (_). The wildcard pattern, like a variable, never fails to match but it entails no binding.

[3] In a local (let) definition, a pattern may occur as the entire left-hand side of an equation. Such an occurrence is implicitly control-disabled, even if it is not prefixed by the character ($\sim$).

[4] As Haskell is strongly typed, a variable can only be compared with terms of the same type.

2.1.2 Constructor patterns: strict and lazy

When a data constructor occurs in a pattern, it must appear in a *saturated* application to sub-patterns. That is, a constructor typed as a k-ary function in a datatype declaration must be applied to exactly k sub-patterns when it is used in a pattern.

When a constructor occurs as the top-level operator in a pattern, a match can occur only if the case discriminator evaluates to a term that has the same constructor as its primary operator. Subterms of the discriminator must match the corresponding sub-patterns of the constructor pattern or else the entire match fails. If a sub-pattern happens to be a variable or a wildcard, no further evaluation of the corresponding sub-term of the matching expression is required.

However, a constructor may be declared (in a datatype declaration) to be *strict* in one or more of its argument positions by prefixing the character (!) to the type expressions in these argument positions. When a constructor is strict in its i^{th} argument position, an application of the constructor will evaluate its i^{th} argument to head normal form. Thus a pattern match involving a constructor declared to be strict in one or more argument positions implicitly forces evaluation of the corresponding subexpressions of a matching term.

2.1.3 Control-disabled patterns

Disabling a pattern for control does not disable the binding function of a match, it merely defers binding until further computation demands a value for one of the variables occurring in the pattern. When that happens, the focus of computation returns to the deferred pattern match, which is fully evaluated in order to bind the variables introduced in the pattern. Should a deferred pattern match fail, no alternative is tried, as might have been the case in a normal match failure. Failure of a deferred pattern match causes an unrecoverable program error.

2.1.4 Matching patterned abstractions

An abstracted pattern fulfills two roles:

- **Control**: A case discriminator expression is evaluated to the extent necessary to determine whether it matches the pattern of a case branch. If the match fails, control shifts to try a match with the next alternative branch, if one is available.
- **Binding**: When a match succeeds, each variable occurring in the pattern is bound to a subterm corresponding in position in the (partially evaluated) case discriminator. Since patterns in Haskell cannot contain repeated occurrences of a variable, the bindings are unique at any successful match.

2.2 Evaluating case expressions

When a case expression is evaluated, the first case branch is applied to the case discriminator (the expression between the keywords case...of). If the

```
-- Semantic Functions for E and P                 -- Environments
mE  :: E -> Env -> V                              type Name = String
mP  :: P -> V -> Maybe [V]                        type Env = Name -> V

-- Domain of Values   {- functions -}      {- structured data -}
          data V =      FV (V -> V)     |     Tagged Name [V]

-- Function composition (diagrammatic)         -- Kleisli composition (diagrammatic)
(>>>) :: (a -> b) -> (b -> c) -> a -> c        (<>) :: (a->Maybe b)->(b->Maybe c)-> a->Maybe c
f >>> g = g . f                                f <> g = \ x -> f x >>= g

-- Domains are pointed                         -- Purification: the "run" of Maybe monad
bottom :: a                                    purify :: Maybe a -> a
bottom = undefined                             purify (Just x) =  x
                                               purify Nothing  = bottom

-- Alternation                                 -- Semantic "seq"
fatbar :: (a->Maybe b) ->                      semseq :: V -> V -> V
             (a->Maybe b) -> (a->Maybe b)      semseq x y = case x of
f `fatbar` g = \ x -> (f x) `fb` (g x)                          (FV _)       -> y ;
    where fb :: Maybe a -> Maybe a -> Maybe a                   (Tagged _ _) -> y
          Nothing `fb` y = y
          (Just v) `fb` y = (Just v)
```

Fig. 2. Semantic Operators

case discriminator matches the abstracted pattern of this branch, then the body of the case branch is evaluated in a context extended with the value bindings of pattern variables made by the match. If the discriminator fails to match the pattern, then the next in the list of case branches is applied to the discriminator, and so on, until a pattern is found to match. If there is no branch whose pattern matches, then evaluation of the case expression fails with an unrecoverable error.

For example, evaluating the following `case` expressions gives these results:

```
data Tree = T Tree Tree | S Tree | L | R
case T L R of {T (S x) y -> y; T x y -> x}     ---> L
case T L R of {T ~(S x) y -> y; T x y -> x}    ---> R
case T L R of {T ~(S x) y -> x; T x y -> y}    ---> program error (match)
case T L R of {~(T (S x) y) -> y; T x y -> x}  ---> program error (match)
```

In the first of the `case` expressions above, the constructor `L` fails to match the embedded pattern `(S x)` in the first case branch. The match failure shifts control to the second case branch. In the second example, the embedded pattern ∼`(S x)` is control-disabled. The term `(T L R)` thus matches the pattern `(T` ∼`(S x) y)` binding `R` to the variable `y`. Since the variable `x` is not demanded, the potential mismatch of pattern `(S x)` with the subterm `L` is never attempted. In the third example, the body of the first case branch demands a value for `x`, thereby forcing a deferred match of the subterm `L` with the pattern ∼`(S x)`. The deferred match fails, resulting in a program error. The fourth example illustrates that a deferred match of the term `(T L R)` against the pattern `(T (S x) y)` fails, although the match was evaluated in response to a request for a binding for `y` alone.

3 Formal Semantics

This section outlines the formal semantics of the Haskell fragment considered in this paper. This semantics has been described in detail elsewhere [3], so the presentation here will be brief. The semantics is presented as a metacircular interpreter for the Haskell fragment whose abstract syntax is specified in Figure 1. The correspondence between terms of the abstract syntax and terms in Haskell's concrete syntax should be evident to the reader with just a few hints. The term `ConApp (‘‘C’’,[Strict,Lazy]) [Var ‘‘x’’,Var ‘‘y]` in the abstract syntax represents a Haskell constructor application `C x y`, where `C` is a data constructor that has been declared strict in its first argument and non-strict in its second argument. A corresponding pattern term in the abstract syntax would be `Pcondata ‘‘C’’ [Pvar ‘‘x’’,Pvar ‘‘y’’]`. Strictness attributes of the data constructor are not explicitly represented in a pattern term.

The interpreter, which is written in Haskell itself, makes use of standard techniques and structures from the denotational description of programming languages and uses a monad to model the control effects of pattern-matching. Although the semantic metalanguage here *is* Haskell, care has been taken to use notation which will be recognizable by any functional programmer. However unlike many functional languages, Haskell has explicit monads, and so we give an overview here of Haskell's monad syntax[5]. The semantics makes use of an error monad [6], which is modeled in Haskell by the `Maybe` monad. This monad is based upon the datatype:

```
Maybe a = Just a  |  Nothing
```

The constructor `Just` encloses a normal expression of type `a` and the constructor `Nothing` models an evaluation failure as a data value.

In every monad there is a unit operator (called `return` in Haskell) and a binary operation which is called "bind" in Haskell and is represented by the infix operator `(>>=)`. The unit of a monad injects an ordinary (non-monadic) value into the structure of the monad. The bind operation is like a function application expressed in diagrammatic order (i.e. *arg* `>>=` *fn*). It accounts for propagation of the monadic structure of the argument through the computation of the application, which may affect that structure. The unit and bind operators satisfy a set of equations that hold for all monads, thus providing a uniform algebraic framework in which to express computational effects.

The structure of Haskell's `Maybe` monad is specified by:

```
data Maybe a = Just a | Nothing
return :: a -> Maybe a              (>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
return = Just                       (Nothing >>= f) = Nothing
                                    (Just x >>= f)  = f x
```

Haskell provides an alternative syntax for bind `(>>=)` called "`do` notation" which is defined by:

[5] We assume the reader has some familiarity with monads [6].

```
mP  :: P -> V -> Maybe [V]
mP (Pvar x) v                         = Just [v]
mP (Pcondata n ps) (Tagged t vs)      = if n==t then (stuple (map mP ps) vs) else Nothing
mP Pwildcard v                        = Just []
mP (Ptilde p) v = Just(case mP p v of { Nothing -> replicate lp bottom ; Just z -> z })
        where lp = length (fringe p)
              replicate 0 x = []
              replicate n x = x : (replicate (n-1) x)

fringe :: P -> [Name]                 --- the fringe of a pattern
fringe (Pvar n)        = [n]          fringe Pwildcard        = []
fringe (Ptilde p)      = fringe p     fringe (Pcondata _ ps) = concat (map fringe ps)

stuple :: [V -> Maybe [V]] -> [V] -> Maybe [V]
stuple [] []         = Just []
stuple (q:qs) (v:vs) = do { v' <- q v ; vs' <- stuple qs vs ; Just (v'++vs') }
```

Fig. 3. Semantics of a Haskell Fragment: Patterns

```
do { y <- x ; f } = (x >>= (\y -> f))
```

Figure 2 contains a description of the semantic setting for the Haskell fragment considered in this paper. It is in most respects a conventional denotational-style semantics for a functional language. The semantics interpretation function for expressions, `mE`, maps an expression and an environment to a value in the domain `V`. The value domain `V` is a c.p.o. with the structure of a *universal domain* [2] over the sum of functions and structured data. The semantics interpretation of patterns is given by the function `mP`, which takes a pattern and returns a map of type `(V -> Maybe [V])` (more will be said about this type in Section 3.1).

We use two composition operators, each written as an infix operator in diagrammatic order. The symbol (`>>>`) denotes function composition and the symbol (`<>`) denotes Kleisli composition in the *Maybe* monad. It is assumed that the domain of values is *pointed* in every type; that is, at every type, the domain contains a bottom element `bottom` (usually written $\perp$), which is modeled in Haskell by the polymorphic constant `undefined`.

Figure 2 also displays two combinators integral to modeling `case` expressions and patterns, called `fatbar` and `purify`. If `m1` and `m2` have type `(V -> Maybe V)`, then `((fatbar m1 m2) v)` exhibits sequencing behavior similar to a `case` expression with two branches, trying first to evaluate the application `m1 v`, then if that fails, trying to evaluate `m2 v`.

The `purify` operator converts a `Maybe`-computation into a value, sending a `Nothing` to `bottom`. Post-composing with `purify` signifies that expressions whose evaluation ultimately result in pattern-match failures (e.g., exhaustion of the branches of a `case` expression) denote `bottom`.

3.1 Dynamic Semantics of the Haskell Pattern Fragment

The dynamic semantics of Haskell `case` expressions (without guards) is given in Figure 4. The semantics of pattern-matching is given in Figure 3. Applying these semantics to the examples given in Section 2.2 yields the indicated

```
mE  :: E -> Env -> V
mE (Var n) rho             = rho n
mE (Case e ml) rho         = mcase rho ml (mE e rho)
mE (ConApp (n,ls) es) rho  = evalL (zip es ls) rho n []
   where evalL :: [(E,LS)] -> Env -> Name -> [V] -> V
         evalL [] rho n vs              = Tagged n vs
         evalL ((e,Strict):es) rho n vs = semseq (mE e rho)
                                                (evalL es rho n (vs ++ [mE e rho]))
         evalL ((e,Lazy):es) rho n vs   = evalL es rho n (vs ++ [mE e rho])
mE Undefined rho      = bottom

match :: Env -> (P,E) -> V -> Maybe V
match rho (p,e) = mP p <> ((\vs -> mE e (extend rho xs vs)) >>> Just)
     where xs = fringe p

mcase :: Env -> [(P,E)] -> V -> V
mcase rho ml = (fatbarL $ map (match rho) ml) >>> purify
                     where fatbarL :: [V -> Maybe V] -> V -> Maybe V
                           fatbarL ms = foldr fatbar (\ _ -> (Just bottom)) ms
```

Fig. 4. Semantics of a Haskell Fragment: Expressions

results.

From the semantics specification we can infer that the effect of deferring pattern match failure is characterized by the following equivalence:
`(mP (~p) v)` is `Just[bottom,...,bottom]` $\Leftrightarrow$ `(mP p v)` is `Nothing`. Even when `(mP p v)` fails (i.e., is `Nothing`), `(mP (~p) v)` still succeeds, but all of the bindings created thereby are `bottom`.

4 Logic

While the dynamic semantics defines a meaning for expressions by providing an abstract evaluation model, a verification logic expresses static assertions about properties of the semantics. An assertion can take the form of a k-ary predicate applied to k terms. For simplicity, we restrict ourselves here to unary predicates ($k = 1$).

We write t ::: P for the assertion that term t satisfies predicate P. Because function and data constructor applications are non-strict in Haskell's evaluation semantics, two notions of satisfaction of a predicate are sensible.

We say that a predicate, P, is *weakly* satisfied by an expression t if the denotation of t belongs to the set specified by P. It is *strongly* satisfied if, in addition, the denotation of t is not the bottom element in its type domain. By convention, a predicate assumes its weak interpretation unless otherwise annotated. An otherwise weak predicate may be explicitly strengthened by prefixing the symbol (\$).

P-logic [6] , a verification logic for Haskell, is based upon the familiar Gentzen-style sequent calculus [1]. In this section we give a brief introduction to the syntax of P-logic, as well as some inference rules that are relevant to pattern-matching in Haskell.

[6] The name P-logic is short for *Programatica logic*, as the logic has been developed as part of the Programatica project [4].

4.1 Predicates in P-logic

Atomic, unary predicates include the predicate constants, Univ and UnDef, which are respectively satisfied by all terms and by only those terms whose values are undefined.

There are two principal ways that compound predicates are formed in *P*-logic.

(i) The constructors of datatypes declared in a Haskell program are implicitly "lifted" to act as predicate constructors in *P*-logic. For example, in the context of a program, the list constructor (:) combines an expression h of type a and an expression t of type [a] into a new expression $(h : t)$ of type [a]. In the context of a formula, the same constructor combines a predicate P and a predicate Q into a new predicate, $(P : Q)$. This predicate is satisfied by a Haskell expression that normalizes to a term of the form $(h : t)$ and whose component expressions weakly satisfy the assertions $h ::: P$ and $t ::: Q$. The default mode of interpretation of the component predicates is weak because the semantics of the data constructor does not require evaluation of its arguments.

(ii) The "arrow" predicate constructor is used to compose predicates that express properties of case branches. An arrow predicate $P \to Q$ is satisfied by a case branch (p -> e) if, whenever the case discriminator satisfies the pattern predicate, P, the body, e, of the case branch satisfies Q.

4.2 Inference Rules for Properties of the Haskell Fragment

4.2.1 Constructor application

Rules for constructor application are derived from a Haskell datatype declaration. A datatype declaration serves to define the data constructors of the type, giving the signature of each constructor as a sequence of type expressions.

$$\mathsf{data}\ T = \cdots \ \mid\ C^{(k)}\,\tau_1 \ldots \tau_k \ \mid\ \cdots$$

Recall that the strictness annotations from the signature of a constructor are represented explicitly in the abstract syntax of a constructor application, although they are not manifested in the concrete syntax.

A constructor application is lifted to a predicate constructor application by the function:

```
sigma :: E -> [Pr] -> Pr
sigma (ConApp (n,ls) _) prs =
   let prs' = take (length ls) prs
       s = and $ map (\(pr,l) -> isStrong pr || l==Lazy) (zip prs' ls)
           where isStrong (Strong _) = True
                 isStrong _ = False
   in if s then Strong $ ConPred (n,ls) prs'
      else ConPred (n,ls) prs'
```

where `ls` lists the strictness declaration (`Lazy` or `Strict`) of the constructor in each argument position. The function zip is defined in Haskell's Standard

```
-- Abstract syntax of predicates
data Pr = Univ | UnDef | PrCon Name [Pr] | Strong Pr

-- Skeleton of a pattern
skeleton :: P -> (Name -> Pr) -> Bool -> (Pr,Bool)
skeleton (Pvar n) e b = case (e n) of
                          pr@(Strong _) -> (pr,True)
                          pr            -> (pr,False)
skeleton Pwildcard e _ = (Univ,False)
skeleton (Pcondata (n,ls) ps) e b =
      let (prs,s) = mapSkel ps e True
      in if b || s then (Strong (PrCon n prs),True)
         else (Univ,False)
skeleton (Ptilde p) e _ = skeleton p e False
```

```
mapSkel :: [P] -> (Name -> Pr)
                 -> Bool -> ([Pr],Bool)
mapSkel [] _ _ = ([],False)
mapSkel (p:ps) e b =
   let (pr,s) = skeleton p e b
       (prs,s') = mapSkel ps e b
   in ((pr:prs),s || s')
```

Fig. 5. Calculation of Pattern Predicates

Prelude as:

$$\mathsf{zip}\,[a_1,\ldots,a_n]\,[b_1,\ldots,b_n] = [(a_1,b_1),\ldots,(a_n,b_n)]$$

Notice that when a predicate constructor lifted from a strict data constructor is applied to a predicate argument, the resulting predicate is strong only if the argument predicate is so, whereas a predicate derived from a lazy data constructor is always strong. A strong predicate formula $\$C^{(k)}\,P_1\ldots P_k$ is satisfied by a well-defined term of the form $C^{(k)}\,t_1\,\ldots\,t_k$ whenever each of the t_j satisfies the corresponding predicate P_j.

Rule schemas for properties of a constructor application are:

(1) $$\frac{\Gamma\vdash_{\mathcal{P}} t_1 ::: P_1 \;\cdots\; \Gamma\vdash_{\mathcal{P}} t_k ::: P_k}{\Gamma\vdash_{\mathcal{P}} C^{(k)}\,t_1\ldots t_k ::: C^{(k)}\,P_1\ldots P_k} \quad (1\le k)$$

and, where C and K are distinct constructors in the same data type:

(2) $$\frac{}{\Gamma\vdash_{\mathcal{P}} C^{(k)}\,t_1\ldots t_k ::: \$\neg K^{(n)}\,\underbrace{\mathsf{Univ}\ldots\mathsf{Univ}}_{n-times}}$$

4.3 Pattern matching

Match clauses have associated with them predicates of a distinct kind. A match clause whose expression body has the Haskell type (τ) may satisfy a predicate of type *Maybe_Pred* τ. These predicates are formed either with the unary predicate constructor $Just$ or the nullary constructor $Nothing$.

4.3.1 Pattern predicates

Because patterns may be nested to arbitrary depths, it is inconvenient to use the syntax of patterns directly in rules. Instead we define a syntactically flattened representation for patterns to allow a simpler representation of pattern predicates in rules.

First, we need the concept of the *fringe* of a pattern, the variables that have defining occurrences in the pattern, listed from left to right.

Definition 4.3.1.1: The *fringe* of a pattern is the list of (distinct) variables whose definition is given by induction on the abstract syntax of patterns by the Haskell function `fringe` in Figure 3.

Next, we define a function that produces a pattern predicate from a pattern and an environment that binds predicates to the variables in the fringe of the pattern.

Definition 4.3.1.2: The *pattern predicate* formed by instantiating a pattern relative to a predicate environment is defined inductively by the Haskell function `skeleton` in Figure 5.
We use the notation $\pi(p)$ to designate a "flattened" pattern predicate constructor. Formally,

$$\pi(p)\ P_1 \cdots P_n =_{def} \mathsf{fst}\ (\mathit{skeleton}\, p\ (\mathsf{zip}\ (\mathit{fringe}\, p)\ [P_1, \ldots, P_n])\ \mathsf{True})$$
$$\text{where } n = \mathit{length}\ (\mathit{fringe}\, p)$$

□

The pattern predicate calculated by *skeleton* will ignore control-disabled subpatterns that occur in a pattern, replacing them by the universal predicate, Univ, *unless* an explicitly strengthened predicate is bound in the environment to a variable in the fringe of such a subpattern. In such a case, the skeleton of the subpattern is fully elaborated in the *skeleton* computation. In consequence, if an instance of a verification rule such as Rule (3) uses strong predicates in its hypotheses, then the pattern predicate in its conclusion will require a pattern match that evaluates all subterms that are asserted by the strong predicates to be well-defined.

For example, two pattern predicates that are derived from one of the patterns given in the example of Section 3.1 are:

$$\pi(\texttt{T}\ \sim\!\texttt{(S x) y})\ \mathsf{Univ}\ \mathsf{Univ} = \$(\texttt{T}\ \mathsf{Univ}\ \mathsf{Univ})$$
$$\pi(\texttt{T}\ \sim\!\texttt{(S x) y})\ \$\mathsf{Univ}\ \mathsf{Univ} = \$(\texttt{T}\ \$(\texttt{S}\ \$\mathsf{Univ})\ \mathsf{Univ})$$

The strength annotation on the first predicate argument in the second line above forces the pattern predicate to assert definedness of the subpattern `(S x)`.

4.3.2 The domain of a pattern

Informally, the *domain* of a pattern is the set of terms that match the pattern. The criterion for matching patterns in Haskell is complicated somewhat by the possibility that a control-disabled subpattern may be embedded into a normally stricter host pattern. In program execution, the match of a control-disabled pattern that is embedded in a case branch is deferred, pending evaluation of the body of the case branch. The match is dynamically performed only if the body is strict in a variable that occurs in the pattern. When a match failure occurs during a deferred pattern match, the match failure is unrecoverable.

We define the domain of a pattern as a predicate characterizing the set of terms matching the pattern in an non-deferred match.

Definition 4.3.2.1: *Dom*(p), is the predicate defined by applying the predicate pattern constructor derived from a pattern, p, to a list of Univ predicates.

$$\mathit{Dom}(p) =_{\mathit{def}} \pi(p)\ \mathsf{Univ} \cdots \mathsf{Univ}$$

The domain predicate of a pattern is calculated by:

```
    dom p = fst (skeleton p (\_ -> Univ) True)
```

□

Notice that *Dom*(p) is either Univ or it is a strong predicate.

The formula $\neg\mathit{Dom}(p)$ asserts that a term fails to match p or is undefined. A strengthened domain predicate disjoined with its strong complement is in effect, a partial definedness predicate. Any term that satisfies either $\mathit{Dom}(p)$ or $\$\neg\mathit{Dom}(p)$ is well defined in every subterm necessary to evaluate a control-enabled match with the pattern p.

4.3.3 Properties of case branches

A case branch has a pair of properties, one that it exhibits when a case discriminator matches its pattern and another that characterizes its behavior when pattern-matching fails:

$$\text{(3)}\qquad \frac{\Gamma,\ x_1 ::: P_1, \cdots, x_n ::: P_n \vdash_{\mathcal{P}} t ::: Q}{\Gamma \vdash_{\mathcal{P}} \{p \texttt{->} t\}\ :::\ \pi(p)\ P_1 \cdots P_n \rightarrow \$\mathit{Just}\ Q}$$

where $[x_1, \ldots, x_n] = \mathit{fringe}\ p$, and

$$\text{(4)}\qquad \Gamma \vdash_{\mathcal{P}} \{p \texttt{->} t\}\ :::\ \$\neg\mathit{Dom}(p) \rightarrow \$\mathit{Nothing}$$

4.3.4 Properties of case expressions

The basic rules for a case expression are those for a single case branch:

$$\text{(5)}\qquad \frac{\Gamma \vdash_{\mathcal{P}} d ::: \pi(p)[P_1, \ldots, P_k] \qquad \Gamma \vdash_{\mathcal{P}} \mathit{br}\ :::\ \pi(p)\ P_1 \cdots P_n \rightarrow \$\mathit{Just}\ Q}{\Gamma \vdash_{\mathcal{P}} \mathsf{case}\ d\ \mathsf{of}\ \{\mathit{br}\} ::: \$\mathit{Just}\ Q}$$

$$\text{(6)}\qquad \frac{\Gamma \vdash_{\mathcal{P}} d ::: \$\neg\mathit{Dom}(p)}{\Gamma \vdash_{\mathcal{P}} \mathsf{case}\ d\ \mathsf{of}\ \{p \texttt{->} t\} ::: \$\mathit{Nothing}}$$

The following rules account for a case expression in which multiple case branches are listed.

$$\text{(7)}\qquad \frac{\Gamma \vdash_{\mathcal{P}} \mathsf{case}\ d\ \mathsf{of}\ \{\mathit{br}\} ::: \$\mathit{Nothing} \qquad \Gamma \vdash_{\mathcal{P}} \mathsf{case}\ d\ \mathsf{of}\ \{\mathit{brs}\} ::: \$Q}{\Gamma \vdash_{\mathcal{P}} \mathsf{case}\ d\ \mathsf{of}\ \{\mathit{br};\ \mathit{brs}\}\ ::: \$Q}$$

where Q has the kind *Maybe_Pred*.

$$\text{(8)}\qquad \frac{\Gamma \vdash_{\mathcal{P}} \mathsf{case}\ d\ \mathsf{of}\ \{\mathit{br}\}\ :::\ \$\mathit{Just}\ P}{\Gamma \vdash_{\mathcal{P}} \mathsf{case}\ d\ \mathsf{of}\ \{\mathit{br};\ \mathit{brs}\}\ :::\ \$\mathit{Just}\ P}$$

Two rules relate a property of a Haskell term of kind *Maybe Pred* to a property of kind *Pred*.

$$\text{(9)}\qquad \frac{\Gamma \vdash_{\mathcal{P}} t\ :::\ \$(\mathit{Just}\ P)}{\Gamma \vdash_{\mathcal{P}} t\ :::\ P} \qquad\qquad \frac{\Gamma \vdash_{\mathcal{P}} t\ :::\ \$\mathit{Nothing}}{\Gamma \vdash_{\mathcal{P}} t\ :::\ \mathsf{Univ}}$$

These rules allow a property of a case expression to be propagated to a context that expects a property of kind *Pred*. When a pattern match can be proven to fail, the concluded property, t ::: Univ, provides no specific information.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash \mathtt{L} ::: \mathsf{Univ}} \qquad \dfrac{}{\vdash \mathtt{R} ::: \$\mathsf{R}}\;(1)}{\vdash (\mathtt{T\,L\,R}) ::: \$(\mathsf{T}\,\mathsf{Univ}\,\$\mathsf{R})}\;(1) \qquad \dfrac{\dfrac{}{\mathtt{x} ::: \mathsf{Univ}, \mathtt{y} ::: \$\mathsf{R} \vdash \mathtt{y} ::: \$\mathsf{R}}}{\vdash \{(\mathtt{T\tilde{}(S\,x)\,y) \text{->} y}\} ::: \$(\mathsf{T}\,\mathsf{Univ}\,\$\mathsf{R}) \rightarrow \$\mathsf{Just}\,\$\mathsf{R}}\;(3)}{\vdash \mathsf{case}\ (\mathtt{T\,L\,R})\ \mathtt{of}\ \{\ (\mathtt{T\tilde{}(S\,x)\,y) \text{->} y}\ \} ::: \$\mathsf{Just}\,\$\mathsf{R}}\;(5)}{\vdash \mathsf{case}\ (\mathtt{T\,L\,R})\ \mathtt{of}\ \{\ (\mathtt{T\tilde{}(S\,x)\,y) \text{->} y}\ \} ::: \$\mathsf{R}}\;(9)$$

$$\dfrac{\dfrac{\dfrac{}{\vdash (\mathtt{T\,L\,R}) ::: \$\neg(\mathsf{S}\,\mathsf{Univ})}\;(2)}{\vdash \mathsf{case}\ (\mathtt{T\,L\,R})\ \mathtt{of}\ \{(\mathtt{S\,x}) \text{->} \mathtt{x}\} ::: \$\mathsf{Nothing}}\;(6)}{\vdash \mathsf{case}\ (\mathtt{T\,L\,R})\ \mathtt{of}\ \{(\mathtt{S\,x}) \text{->} \mathtt{x}\} ::: \mathsf{Univ}}\;(9)$$

Fig. 6. Distinguishing Pattern Match Success from Failure in the Logic
(Numbers refer to the rule that applies at each step.)

Figure 6 presents a sample derivation demonstrating how P-logic distinguishes pattern-matching success and failure. The second proof involves a case expression which generates a pattern match failure. The only property derivable of this expression in P-logic is Univ.

5 Conclusion

We have presented two succinct formalisms to specify the reduction semantics of Haskell pattern-matching, a surprisingly complex aspect of the language. A denotational semantics furnishes an abstract model for computation in the language. A verification logic provides a deduction system in which to state and prove properties of computations. The dual development of a denotational semantics and a verification logic for a programming language affords the opportunity to check soundness of the logical inference rules relative to the model provided by the semantics. When the semantics is also given in an executable framework, as has been done here for Haskell, soundness checking can be partially automated, but a discussion of technique topic is left to another paper.

We have focused attention on the fragment of Haskell in which expression evaluation is directed by pattern-matching, as that aspect of the language has seemed most in need of formal characterization. The deferred matching that is required of control-disabled ($\sim$) patterns is of particular interest. In the dynamic semantics, a deferred pattern match is embedded in a continuation that substitutes the value `bottom` in the bindings of pattern variables in case the match fails. Thus any reference to one of those variables occurring in the expression part if the case branch headed by that pattern is a reference to the value `bottom`. In a logical characterization of a deferred pattern match, unless it is explicitly asserted that one or more of the pattern variables satisfies a strong property, only the property Univ is required of an argument in a deferred pattern match. Thus, nothing is assumed about values taken by pattern variables, which is compatible with the binding of `bottom` values specified in the semantics.

In retrospect, one might question whether the generalization of control-disabled patterns in Haskell, allowing ($\sim$) annotations to be embedded in a pattern in any context, has been worth the complexity that it adds to intuitively understanding the semantics of Haskell. For this language feature, the logical characterization actually aids intuition, as well as providing rules for formal reasoning.

Acknowledgment The authors wish to thank their colleagues on the Programatica project, particularly Jim Hook, Mark Jones and Sylvain Conchon for their encouragement and for numerous discussions on aspects of logic and Haskell semantics.

References

[1] Jean-Yves Girard. *Proofs and types*, volume 7 of *Cambridge tracts in theoretical computer science.* Cambridge University Press, 1989.

[2] Carl A. Gunter. *Semantics of Programming Languages: Programming Techniques.* The MIT Press, Cambridge, Massachusetts, 1992.

[3] William Harrison, Tim Sheard, and James Hook. Fine control of demand in haskell. In *Sixth International Conference on the Mathematics of Program Construction, Dagstuhl, Germany*, volume 2386 of *Lecture Notes in Computer Science*, pages 68–93. Springer Verlag, July 2002.

[4] Programatica Home Page. `www.cse.ogi.edu/PacSoft/projects/programatica`. James Hook, Principal Investigator.

[5] Simon Peyton Jones and editors John Hughes. Report on the programming language Haskell 98. Technical Report YALEU/DCS/RR-1106, Yale University, CS Dept., February 1999.

[6] P. Wadler. The essence of functional programming. *19th POPL*, pages 1–14, January 1992.