Reflecting Demand in Programming Logics: a case study for Haskell

William L. Harrison Richard B. Kieburtz

Pacific Software Research Center
OGI School of Science & Engineering
Oregon Health & Science University
{wlh,dick}@cse.ogi.edu

Abstract. Haskell is a functional programming language whose evaluation is lazy by default. However, Haskell also provides pattern matching facilities which add a modicum of eagerness to its otherwise lazy default evaluation. This mixed or "non-strict" semantics can be quite difficult to reason with. This paper introduces a programming logic, *P*-logic, which neatly formalizes the mixed evaluation in Haskell pattern-matching as a logic, thereby simplifying the task of specifying and verifying Haskell programs. In *P*-logic, aspects of demand are reflected or represented within both the predicate language and its model theory, allowing for expressive and comprehensible program verification.

1 Introduction

Haskell is frequently referred to as a "lazy" functional language, but it is more properly understood as *non-eager* language because Haskell contains many constructs which perturb its default lazy evaluation. Among these perturbing features are pattern-matching, datatype strictness annotations, and irrefutable pattern annotations, and their combination in Haskell yields some of the most sophisticated pattern-matching facilities around. The interaction between Haskell's default lazy evaluation and its pattern-matching is surprisingly complicated[9] and has given rise to the slogan: *Haskell has fine control of demand.*

This paper introduces an axiomatic semantics for Haskell, called *P*-logic[1], which neatly characterizes Haskell's fine control of demand in a logic. *P*-logic is based upon the familiar Gentzen-style sequent calculus [4]. Predicates in *P*-logic have two interpretations—weak and strong—that are used to mimic lazy and eager demand in some sense. Within the predicate language itself, predicates may be explicitly *strengthened* to coerce a predicate's interpretation to be strong. Verification conditions for predicates—called *pattern predicates*—are calculated directly from each pattern, and these verification condition generators[2] are used throughout the inference rules presented here. The verification condition generators are the mechanism by which demand in Haskell is reflected within *P*-logic.

The rest of this paper proceeds as follows. First, Section 2 gives an overview of the Haskell sublanguage we consider in this paper. This sublanguage contains the Haskell constructs most directly affected by its pattern-matching. Section 3 presents an overview of the formal semantics of this fragment. Although this semantics has been presented elsewhere[9], we include a summary here to make the paper more self-contained and also because this semantics influenced the design of *P*-logic. Finally in Section 4, we present the fragment of *P*-logic which deals with Haskell's fine control of demand.

[1] The name *P*-logic is short for *Programatica logic*, as the logic has been developed as part of the Programatica project [15] at OGI.

[2] Written in Haskell and available by request from the authors.

2 A Haskell fragment and its informal semantics

This section gives an overview of the fragment of Haskell we consider in this paper. The Haskell sublanguage (expressed as the Haskell datatype `E` in Figure 1) is representative of the Haskell constructs dependent on pattern-matching. First in Section 2.1,we give an informal overview of the meaning of these constructs, and then in Section 2.2, we show how Haskell's fine control of demand leads to unexpected complexity in Haskell evaluation.

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

Fig. 1. Abstract Syntax of a Haskell Fragment

2.1 Patterned abstractions

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 in `case` expressions.

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 case discriminator matches the abstracted pattern of the 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. If no branch matches, then evaluation of the case expression fails with an unrecoverable error.

Matching abstracted patterns 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 (partly evaluated) `case` discriminator. Since patterns in Haskell cannot contain repeated occurrences of a variable, the bindings are unique at any successful match.

[3] In a local 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 (∼).

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.

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 its i^{th} argument position, a constructor application will evaluate its i^{th} argument. 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 the matching term.

Control-disabled patterns Disabling a pattern for control with (~) 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 computed 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.

2.2 An example of Haskell's "Fine Control of Demand"

In the evaluation of (`case` e $\{p_1$ `->` $e_1; \ldots ; p_n$ `->` $e_n\}$), patterns p_i are matched against e in left-to-right order until a successful match p_s is found. Then, the value of the whole expression is the value of the expression e_s. If no such match is found, then the value of the whole expression is undefined. For example, consider the following `case` expressions in Table 1:

```
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)
```

Table 1. "Fine control of demand" in Haskell complicates evaluation

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

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

```
-- Semantic Functions for E and P   -- Environments        -- Domain of Values
mE  :: E -> Env -> V                type Name = String     data V = FV (V -> V) |     {- functions -}
mP  :: P -> V -> Maybe [V]          type Env = Name -> V           Tagged Name [V]   {- structured data -}

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

Fig. 2. Semantic Operators

the pattern `(T ~(S x) y)` binding `R` to the variable `y`. 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 of the Haskell Fragment

This section outlines the formal semantics of the Haskell fragment considered in this paper. This semantics has been described in detail elsewhere [9], and so the presentation here will be brief. The semantics is presented as a metacircular interpreter for the Haskell fragment whose abstract syntax is given in Figure 1. The interpreter, written in Haskell itself, makes use of standard techniques and structures from the denotational description of programming languages and uses monads to model 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 relies on an error monad [18], which is modeled in Haskell by the `Maybe` monad. The structure of the `Maybe` monad, its unit (`return`) and its bind `(>>=)`[6] are given as:

```
data Maybe a = Just a | Nothing       (>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
return :: a -> Maybe a                (Nothing >>= f) = Nothing
return = Just                         (Just x >>= f)  = f x
                                      do { y <- x ; f } = (x >>= (\y->f))
```

Lastly, 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 1 (Fringe of a pattern) *The* fringe *of a pattern p is the list of (distinct) variables occurring in p in left-to-right order. Its definition is given by induction on the abstract syntax of patterns by the Haskell function* `fringe` *in Figure 3.*

[5] We assume the reader has some familiarity with monads [18].

[6] Haskell has an alternative syntax for bind `(>>=)` called "`do` notation" which is defined above.

```
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

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 semantics for a functional language. The dynamic semantics for expressions, `mE`, maps an expression and an environment to a value in the domain of values `V`. This domain `V` is itself structured conventionally as a universal domain construction [8] over the sum of functions and structured data. The semantics of patterns, `mP`, takes a pattern and returns a map of type `(V->Maybe [V])` (more will be said about this type below). We use two composition operators, both written as infix operators in diagrammatic order. The symbol (`>>>`) denotes function composition and the symbol (`<>`) denotes Kleisli composition in the *Maybe* monad. It is assumed that the domain is *pointed* in every type; that is, every domain contains a bottom element `bottom` (usually written $\perp$), which is modeled here by the polymorphic Haskell 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 `(case v of { m1 ; m2 })`. The `purify` operator converts a `Maybe`-computation into a value, sending a `Nothing` to `bottom`. Post-composing with `purify` signifies that expressions whose evaluation produces certain pattern-match failures (e.g., exhaustion of the branches of a `case` expression) ultimately denote `bottom`.

Figures 3 and 4 display the semantics for patterns and expressions, `mP` and `mE`, respectively. For a full description of these semantics, please refer to [9]. Generally, 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 for the Haskell Fragment

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.

```
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

Definition 2 (Weak/Strong Predicate Satisfaction) *We say that a predicate, P, is* weakly satisfied *by an expression t of type τ if t's denotation belongs to the set specified by P. It is* strongly satisfied *if, in addition, the denotation of t is not the bottom element in the domain corresponding to the type τ. By convention, a predicate assumes its weak interpretation unless otherwise annotated. An otherwise weak predicate may be explicitly strengthened by prefixing the symbol (\$).*

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.

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.

1. 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.
2. The "arrow" predicate constructor is used to compose predicates that express properties of case branches. An arrow predicate $P \rightarrow Q$ 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.

Figure 5 contains a Haskell definition of the abstract syntax for the P-logic predicate language.

```
data Pr = Univ                      -- the Universal predicate
        | UnDef                     -- the Undefined predicate
        | ConPred (Name,[LS]) [Pr]  -- a pattern predicate
        | Strong Pr                 -- a strengthened predicate
        | Predconst Integer         -- a predicate asserting a constant value
        | PredVar Name              -- a predicate variable
        | PArrow Pr Pr              -- arrow predicates
data LS = Lazy | Strict deriving Eq
```

Fig. 5. Abstract syntax of predicates as Haskell datatype

4.2 Inference Rules for Properties of the Haskell Fragment

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.

$$\texttt{data}\ T = \cdots\ \texttt{|}\ C^{(k)}\ \tau_1 \ldots \tau_k\ \texttt{|}\ \cdots$$

Notice 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:

```
conApp :: E -> [Pr] -> Pr
conApp (ConApp (n,ls) es) 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. 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.

The rule schemas for properties of a constructor application are:

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

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

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

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`.

```
    {---  the pattern predicate   ---}     pat2pred :: P -> [Pr] -> Pr
                                           pat2pred p plist = fst (patPred p (zip (fringe p) list) True)

    {--- pattern domain predicate ---}     dom p = pat2pred p (replicate pl Univ)
                                                      where pl = length (fringe p)

              patPred :: P -> [(Name,Pr)] -> Bool -> (Pr,Bool)
              patPred (Pvar x) sigma b   =
                    (purify (lookup x sigma), is_strong (purify (lookup x sigma)))
                               where
                                is_strong (Strong _) = True
                                is_strong _          = False

              patPred Pwildcard sigma b       = (Univ,False)
              patPred (Ptilde p) sigma b      = patPred p sigma False
              patPred (Pcondata n ps) sigma b =
                     let (ConPred "List" preds, s) = map_patPred ps sigma b
                  in
                     if (s || b) then
                        (Strong (ConPred n preds), True)
                 else   (Univ, False)

              map_patPred :: [P] -> [(Name,Pr)] -> Bool -> (Pr,Bool)
              map_patPred [] sigma b     = (ConPred "List" [], False)
              map_patPred (p:ps) sigma b = (ConPred "List" (pp:pps), s1 || s2)
                                 where
                                     (pp,s1)                 = patPred p sigma b
                                     (ConPred "List" pps,s2) = map_patPred ps sigma b
```

Fig. 6. Calculation of Pattern Predicates

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.

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. Note that

$$\texttt{fst}(x, y) = x \text{ and } \texttt{zip}\,[a_1, \ldots, a_n]\,[b_1, \ldots, b_n] = [(a_1, b_1), \ldots, (a_n, b_n)]$$

Definition 3 (Pattern predicate) *The* pattern predicate *formed by instantiating a pattern relative to a predicate environment is defined inductively by the Haskell function* `pat2pred` *in Figure 6. We use the notation $\pi(p)$ to designate a "flattened" pattern predicate constructor.*

The pattern predicate calculated by `pat2pred` 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 `pat2pred` 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 examples of Table 1 are:

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

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)`.

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 (Pattern Domain Predicate) *The* domain predicate of pattern *p,$Dom(p)$, is the predicate defined by applying the predicate pattern constructor derived from a pattern, p, to a list of* Univ *predicates.*

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

The domain predicate of a pattern is calculated by the Haskell function `dom` *shown in Figure 6.*

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

The formula $\neg Dom(p)$ 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 $Dom(p)$ or $\$\neg Dom(p)$ is well defined in every subterm necessary to evaluate a control-enabled match with the pattern p.

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:

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

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

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

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

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

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

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

$$\frac{\Gamma \vdash_{\mathcal{P}} \texttt{case}\ d\ \texttt{of}\ \{br\} ::: \$\texttt{Nothing} \qquad \Gamma \vdash_{\mathcal{P}} \texttt{case}\ d\ \texttt{of}\ \{brs\} ::: \$Q}{\Gamma \vdash_{\mathcal{P}} \texttt{case}\ d\ \texttt{of}\ \{br;\ brs\}\ :::\ \$Q} \tag{7}$$

where Q has the kind *Maybe_Pred*.

$$\frac{\Gamma \vdash_{\mathcal{P}} \texttt{case}\ d\ \texttt{of}\ \{br\}\ :::\ \$\texttt{Just}\ P}{\Gamma \vdash_{\mathcal{P}} \texttt{case}\ d\ \texttt{of}\ \{br;\ brs\}\ :::\ \$\texttt{Just}\ P} \tag{8}$$

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

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

Fig. 7. Distinguishing Pattern Match Success from Failure in the Logic
(Numbers refer to the rule that applies at each step.)

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

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

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.

5 Related Work

As part of the *Programatica*[15] project at the Pacific Software Research Center, we are attempting to develop both a formal basis for reasoning about Haskell programs, and automated tools for mechanizing such reasoning. An important part of our work is to develop a logic with which to manipulate Haskell terms.

Simon Thompson's early effort to give a verification logic [17] for Miranda, a lazy, functional language that was a predecessor to Haskell, exposed many of the difficulties inherent in adapting a first-order predicate calculus for use as a verification logic. The logic for Miranda employs quantification operators that bind variables to range only over *defined* terms, or over *finite* structures of a datatype. The meanings of such quantifiers are extra-logical; they cannot be defined in the logic itself.

Sparkle [2] is a verification tool for *Clean* [14], a lazy functional programming language. *Sparkle* is a tactical theorem prover for a first-order logic, specialized to verifying properties of functional programs. Expressions of the term language, *Core-Clean*, can be embedded in propositions, including logical variables bound by universal or existential quantifiers. The *Sparkle* logic has a notation to express an undefined value but does not provide modalities.

In formulating *P*-logic, we are interested in models constructed of the unbounded terms of a specific abstract syntax. From the *Stratego* language[16] we learned of data constructor congruences, whereby the initial algebra property of a freely constructed datatype is used to lift strategies for rewriting the arguments of a particular construction into a homomorphic strategy for rewriting the construction itself. In *P*-logic, constructor congruences are used to form homomorphic predicates satisfied by constructed terms from predicates that characterize subterms.

A different kind of modality is used in *P*-logic to characterize normalization of terms by differentiating strong and weak satisfaction criteria. The introduction of this modality was inspired by a three-valued propositional logic, *WS*-logic [13], which conservatively extends classical propositional logic, with the notable exception that the trivial sequent, $P \vdash P$ is not sound.

A modality analogous to the *weak—strong* modality of P-logic was introduced by Larsen [12] to discriminate *must* and *may* transitions in a process algebra. He observed that conventional process models specify only *may*, or nondeterministic, transitions and therefore, only safety properties can be stated of such a model. By introducing *must*, or required transitions, it is also possible to assert liveness properties.

Huth, Jagadeesan and Schmidt [10] generalized Larsen's analysis and provided a semantic interpretation of the modality in a more general framework. Their semantic interpretation of a predicate is a pair of power-domain elements, $(P_\perp, P_\top)$, where $P_\perp$ is downward-closed and $P_\top$ is upward-dense. These interpretations are used in modeling *may* and *must* properties, respectively. This general characterization of predicate interpretations also applies to the weak and strong notions of predicate satisfaction that we have used in P-logic.

All programming logics must confront the issue of undefinedness because all programming languages admit programs which are undefined for some inputs. Among the sources of such undefinedness are non-termination, pattern-matching failure, arithmetic errors (e.g., division by zero), etc. Partial logics—logics that deal with undefinedness—have been studied intensely for years as a basis for programming logics. A far from complete list includes [13, 6, 7, 1, 3, 5, 11]. For an excellent overview, the interested reader should consult Farmer[3].

6 Conclusion

Figure 7 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. Significantly, the strongest property derivable of this expression in P-logic is also the one which contains no information at all: namely, `Univ`.

We have presented a two succinct formalisms that specify the dynamic and axiomatic semantics of Haskell pattern-matching, a surprisingly complex aspect of the language. The deferred matching that is required of control-disabled ($\sim$) patterns is of particular interest. In the dynamic semantics, expressed as a meta-language program in Haskell, 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. In the verification logic, a pattern predicate is calculated from the pattern and the predicates assumed to be necessary to prove a property of the body of a case branch. This also ensures that a proof of properties will discriminate between failure of a deferred match and failure of a normal match.

P-logic is a verification logic for all of Haskell98, although we have only shown the part dealing with Haskell's fine control of demand here. The standard interpretation of P-logic and its soundness with respect to this model have been demonstrated, although it is beyond the scope of the current paper to describe them here.

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

References

1. Jen H. Cheng and Cliff B. Jones. On the usability of logics which handle partial functions. In C. Morgan and J. C. P. Woodcock, editors, *Proceedings of the Third refinement Workshop*, Workshops in Computing Series, pages 51–69, Berlin, 1991. Springer-Verlag.
2. Maarten de Mol, Marko van Eekelen, and Rinus Plasmeijer. Theorem proving for functional programmers. In *Proceedings of the 13th International Workshop on the Implementation of Functional Programming Languages (IFL'01)*, pages 99–118, September 2001.

3. William M. Farmer. Reasoning about partial functions. *Erkenntnis*, 43:279–294, 1995.
4. Jean-Yves Girard. *Proofs and types*, volume 7 of *Cambridge tracts in theoretical computer science*. Cambridge University Press, 1989.
5. David Gries and Fred B. Schneider. Avoiding the undefined by underspecification. In Jan van Leeuwen, editor, *Computer Science Today: Recent Trends and Developments*, number 1000 in Lecture Notes in Computer Science, pages 366–373. Springer-Verlag, New York, NY, 1995.
6. Raymond D. Gumb and Karel Lambert. A free logical foundation for nonstrict functions. In *Proceedings of the CADE-13 Workshop on the Mechanization of Partial Functions*, pages 39–46, 1996.
7. Raymond D. Gumb and Karel Lambert. Definitions in nonstrict positive free logic. *Modern Logic*, 7:25–55, 1997.
8. Carl A. Gunter. *Semantics of Programming Languages: Programming Techniques.* The MIT Press, Cambridge, Massachusetts, 1992.
9. William Harrison, Timothy Sheard, and James Hook. Fine control of demand in haskell. In *6th International Conference on the Mathematics of Program Construction, Dagstuhl, Germany*, volume 2386 of *Lecture Notes in Computer Science*, pages 68–93. Springer-Verlag, 2002.
10. Michael Huth, Radha Jagadeesan, and David Schmidt. Modal transition systems: A foundation for three-valued program analysis. *Lecture Notes in Computer Science*, 2028, 2001.
11. B. Konikowska, A. Tarlecki, and A. Blikle. A three-valued logic for software specification and validation. *Fundamenta Informaticae*, XIV:411–453, 1991.
12. K. G. Larsen. Modal specifications. In J. Sifakis, editor, *Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems*, volume 407 of *LNCS*, pages 232–246, Berlin, June 1990. Springer.
13. Olaf Owe. Partial logics reconsidered: A conservative approach. *Formal Aspects of Computing*, 5(3):208–223, 1993.
14. Rinus Plasmeijer and Marko van Eekelen. Functional programming: Keep it CLEAN: A unique approach to functional programming. *ACM SIGPLAN Notices*, 34(6):23–31, June 1999.
15. Programatica Home Page. `www.cse.ogi.edu/PacSoft/projects/programatica`. James Hook, Principal Investigator.
16. Stratego Home Page. `www.stratego-language.org`.
17. Simon Thompson. A Logic for Miranda, Revisited. *Formal Aspects of Computing*, 7:412–429, 1995.
18. Phillip Wadler. The essence of functional programming. *19th POPL*, pages 1–14, January 1992.