Fine Control of Demand in Haskell*

William Harrison, Tim Sheard, and James Hook

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

Abstract. Functional languages have the λ-calculus at their core, but then depart from this firm foundation by including features that alter their default evaluation order. The resulting *mixed* evaluation—partly lazy and partly strict—complicates the formal semantics of these languages. The functional language Haskell is such a language, with features such as pattern-matching, case expressions with guards, etc., introducing a modicum of strictness into the otherwise lazy language. But just how does Haskell differ from the lazy λ-calculus? We answer this question by introducing a calculational semantics for Haskell that exposes the interaction of its strict features with its default laziness.

1 Introduction

"Real" functional programming languages are neither completely lazy nor completely strict—rather, they are a mixture of the two. Functional languages commonly contain constructs which alter the default evaluation strategy to make programs more efficient or useful in some respect. So-called strict languages like ML[9] and Scheme[1, 3] both contain a lazy *if-then-else*, without which programming in them would be much more difficult. Haskell[13] is rife with features perturbing its default lazy evaluation to allow control of demand (primarily for reasons of efficiency). Now at the heart of real functional languages lie various flavors of the λ-calculus (i.e., strict and lazy), and the semantics of these have been well-understood[5, 19] for some time. But just how do the semantics of real languages with messy, mixed evaluation relate to these textbook examples? In this paper, we answer this question for a large fragment of Haskell[13].

The contributions of the present work are three-fold. We demonstrate techniques for rigorously specifying "mixed" languages. Many of these techniques—particularly the development of nested patterns—apply to the semantics of functional languages in general. We provide a large case study of the development of a formal language specification from an informal one. The Haskell98 report[13] describes Haskell in a semi-formal manner, and in many instances, these descriptions guided the development of the formal specification. Finally, this work

* The work described here was supported in part by National Science Foundation Grant CDA-9703218 and the M.J. Murdock Charitable Trust.

presents a dynamic semantics for a larger fragment of Haskell (including essentially everything but overloading[6, 7]) than has ever been gathered together in one place before.

Haskell contains a large number of constructs allowing some control over evaluation and, thus, departing from its standard lazy evaluation. We say that such constructs introduce "fine control of demand," because they all make subtle changes to the default lazy evaluation strategy of Haskell, and all for compelling reasons. The Haskell features with fine control of demand are: nested patterns, `case` expressions and `let` declarations, guards and `where` clauses on equations, strict constructor functions, the `newtype` datatype definition facility, and the `seq` operator. This paper presents a model for all of these features.

In this paper we present a *calculational* semantics which provides a unified model of these constructs and their interactions with Haskell's other features. By *calculational* semantics, we mean a meta-circular interpreter[1] for Haskell, written in Haskell. We distinguish the *calculational* approach to language definition taken in the present work from a denotational one, although the two are similar in form and spirit. The semantics presented here is a Haskell program, but it has been designed using standard techniques and structures from programming language semantics.

The challenge we confronted immediately in writing this language specification arose from the fact that these evaluation control constructs interact with one another in ways that were difficult to understand. Because of this interaction, it was very easy to write a language definition that was either too lazy or too strict. Automated support (in the form of type-checking and unit testing of specifications) was very helpful in eliminating bugs and establishing confidence in the correctness of the definition.

An alternative to the calculational approach to defining languages advocated here would be to provide a purely formal mathematical specification (i.e., a set of equations on paper). But writing the semantics as a functional program has a number of advantages over this approach. The language specification presented here is both type-checked and executable. Subtle errors may be caught and more easily corrected in a type-checked environment. The interpreter presented here has been tested on a large number of examples. Unit testing was invaluable in identifying problems in earlier versions of this semantics, and such problems may well have been overlooked in a purely mathematical specification. Indeed, several earlier, purely formal mathematical specifications for nested pattern matching proved either too lazy or too strict, and having an executable version helped us to expose and isolate the subtleties involved.

1.1 Rationale

As part of the *Programatica*[18] 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, and a standard model is required to establish the soundness of this logic. This

led us to the literature, which to our surprise, was lacking in formal descriptions of the *big picture* of Haskell. There are plenty of papers about particular features of Haskell, (its laziness[12], its class system[7], etc.) but very little work which unifies the *essence of Haskell* with all the fine-control mechanisms that have been added and refined over the years.

The Haskell98 report[13] uses a strategy of translating many complex constructs into a simpler core language[14], that is really just a slightly extended lambda-calculus. However, the translation-based approach, while useful and intuitive, is problematic as a semantic definition (we expand on these issues in Appendix A). Some of these translations are semantically harmless, amounting to nothing more than the removal of so-called "syntactic sugar." However, many of the translation schemas in the Haskell Report rely on the generation of new source-level variables. The semantics of languages with variable binding mechanisms are very complex, and when one defines a language feature by a translation that introduces new variables, one leaves the well-understood semantic setting of domains and continuous functions, and moves into a much more complicated world. Recent work on languages with binding mechanisms suggests that these kinds of transformations are by no means trivial to model correctly[10, 16]. Another serious defect with the translation-based approach is that it fails to be compositional. For this reason we eschew such techniques when there are relatively simple alternatives we can adapt from denotational semantics. However, many of the specifications presented here are inspired by the translations given in the Haskell98 report, and we compare our definitions with those of the report when appropriate.

Although the semantic metalanguage here *is* Haskell, care has been taken to use notation which will be recognizable by any functional programmer. However unlike the languages ML[9], Miranda[11], Scheme[1, 3], and Clean[17], Haskell does have built-in monads, and so we give an overview here of Haskell's monad syntax[1]. The semantics relies on an error monad[8], which is built-in to Haskell as the `Maybe` monad. The structure of the `Maybe` monad, its unit `return`, and its bind `(>>=)` 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))
```

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

2 The Abstract Syntax

Haskell is a rich language with many features, and the sublanguage presented here identifies 9 different syntactic categories. These categories include names,

[1] We assume the reader has some familiarity with monads[20, 8].

```
type Name = String
data Op = Plus | Mult | IntEq | IntLess
data LS = Lazy | Strict deriving Eq

data P --- Nested, Linear (i.e., no repetition of variables) Patterns
  = Pconst Integer          -- { 5 }
  | Pvar Name               -- { x }
  | Ptuple [P]              -- { (p1,p2) }
  | Pcondata Name [P]       -- data T1 = C1 t1 t2; {C1 p1 p1} = e
  | Pnewdata Name P         -- newtype T2 = C2 t1;  {C2 p1} = e
  | Ptilde P                -- { ~p }
  | Pwildcard               -- { _ }

data E --- Haskell Expressions
  = Var Name                -- { x }
  | Const Integer           -- { 5 }
  | Undefined               -- { undefined }
  | App E E                 -- { f x }
  | Abs [P] E               -- { \ p1 p2 -> e }
  | TupleExp [E]            -- { (e1,e2) }
  | ConApp Name [(E,LS)]    -- data T1 = C1 t1 t2; p = {C1 e1 e2}
  | NewApp Name E           -- newtype T2 = C2 t1; p = {C2 e1}
  | Seq E E                 -- { seq e1 e2 }
  | Bin Op E E              -- { e1 + e2 }
  | Cond E E E              -- { if e1 then e2 else e3 }
  | Let [D] E               -- { let x=e1; y=e2 in e3 }
  | Case E [Match]          -- { case e of p -> b where ds ; ... }

type Match  = (P,B,[D])     -- case e of { pat -> body where decs }
type Clause = ([P],B,[D])   -- f { p1 p2 = body where decs }

data D --- Declarations
  = Fun Name [Clause]       -- f p1 p2 = b where ds
  | Val P B [D]             -- p = b where ds

data B -- Bodies
  = Guarded [(E,E)]         -- f p { | e1 = e2 | e3 = e4 } where ds
  | Normal E                -- f p = { e } where ds
```

Fig. 1. Abstract Syntax of Haskell Sublanguage

operators, strictness annotations, matches, clauses, bodies, expressions, declarations, and patterns. In Figure 1 we display the `data` definitions in Haskell that represent our abstract syntax. Our definitions are conventional and we have used comments to relate the abstract syntax to the concrete syntax of Haskell. The missing syntax, necessary to complete a Haskell definition, has mostly to do with the module system, classes, list comprehensions, and the do notation.

```
-- Scalar, function, and structured data values
   data V = Z Integer  | FV (V -> V) | Tagged Name [V]
-- Environments bind names to values
   type Env = Name -> V
-- Meanings for expressions, patterns, bodies, and declarations
   mE  :: E -> Env -> V                mB  :: B -> Env -> Maybe V
   mP  :: P -> V -> Maybe [V]          mD  :: D -> Env -> V
```

Fig. 2. Calculational Semantics of Haskell

3 A Model for Haskell

This section presents a calculational model of Haskell as a compositional metacircular interpreter, and Figure 2 presents the definitions of the semantic values `V`, environments `Env`, and the types of the semantic functions for expressions (`mE`), patterns (`mP`), bodies (`mB`), and declarations (`mD`). Semantic values are constructed in a standard way, corresponding to a universal domain construction[5] in denotational semantics. Environments map names to values. All of the semantic functions are *compositional*: the meaning of any term from the syntactic categories `E`, `P`, `B`, and `D` depends solely on the meaning of its subterms.

We model laziness in Haskell using the laziness of the metalanguage (i.e., Haskell) and to some degree this limits the generality of the semantics presented here. An alternative we have explored is a *Reynolds-style* monadic interpreter[2] which models laziness explicitly. The advantage of this Reynolds-style approach is that the resulting semantics could be executed in any functional language—even strict languages like ML and Scheme. But the disadvantage is the unavoidable loss of precision in the typing of the semantics. In a Reynolds-style monadic interpreter, all semantic functions are monadically-typed, and it is difficult in such a setting to distinguish clearly between the "pure" value world of expressions and declarations (i.e., `mE` and `mD`) and the computational world of patterns and bodies (i.e., `mP` and `mB`) as we do here.

3.1 Modeling failure

In Haskell, there are several distinct kinds of failure, only one of which is explicitly modeled by the `Maybe` monad.

1. The first kind of failure arises from run-time errors, such as division by zero, and non-exhaustive pattern match coverage. Taking the `head` of the empty list is an example of this kind of failure.
2. The second kind of failure stems from non-termination. This kind of failure is captured by the interpreter itself not terminating.
3. The third kind of failure stems from an unsuccessful pattern match in a context where the failure may be trapped and controlled by proceeding to the next match of a case expression or the next clause of a multi-line function definition. We model this failure as a computation in the `Maybe` monad. Such a trappable failure can become a failure of the first kind, if it occurs in a context with no next match or clause.

In our semantics, a failure of the first kind is not reported—it causes the interpreter to terminate unsuccessfully. We assert that programs which fail in the manner of (1.) and (2.) above denote `bottom` (where `bottom` is the semantic value usually written "$\perp$"—see Figure 3 below).

3.2 Semantic Operators

Rather than give an explicitly categorical or domain-theoretic treatment here, we summarize our assumptions as the existence of certain basic semantic operators (shown in Figure 3). The first three operations, function composition `>>>`, function application `app`, and currying `sharp`, are basic operations in denotational descriptions of functional programming languages. A call to the currying operation `(sharp n [] beta)` converts an uncurried function value `beta` of the form (`\[`$v_1, \ldots, v_n$`]->`$body$) into an equivalent curried form (`\`v_1`->` $\ldots$ `\`v_n`->`$body$). We also assume that each semantic domain corresponding to a Haskell type contains a bottom element `bottom`—that is, that each domain is *pointed*. This is necessary for the existence of least fixed points, which are themselves necessary for modeling recursion. We define a least fixed point operator, `fix`, in the standard way.

The `semseq` operation requires some explanation. The operation `semseq` is meant to specify the Haskell operation `seq`, and it is named accordingly (for "*semantic seq*"). The purpose of `seq` is to allow Haskell programmers to force evaluation to occur, and the benefit of `seq` is only evident in a call-by-need semantics with its attendant sharing. In such a model, one uses `(seq x y)` to force computation of the first operand, so that subsequent evaluations of `x` will use its shared value.

In the Haskell98 report[13] (cf. Section 6.2, page 75), `(seq :: a->b->b)` is defined by the following equations:

$$\texttt{seq}\ \perp\ y = \perp \text{ and } \texttt{seq}\ x\ y = y, \text{ if } x \neq \perp$$

The operator `(semseq x y)` is defined similarly, although with a subtlety arising from our model of failure. `(semseq x y)` evaluates its first operand `x` sufficiently to match it to a value in `V` and, in doing so, may produce a failure of either forms (1) or (2) listed above in Section 3.1 (and thus ultimately producing `bottom`).

```
-- Function composition (diagrammatic) & application
(>>>) :: (a -> b) -> (b -> c) -> a -> c
f >>> g = g . f
app :: V -> V -> V
app (FV f) x = f x

-- Currying
sharp :: Int -> [V] -> (V -> V) -> V
sharp 0 vs beta = beta (tuple vs)
sharp n vs beta = FV $ \ v -> sharp (n-1) (vs++[v]) beta
    where tuple :: [V] -> V
          tuple [v] = v
          tuple vs = Tagged "tuple" vs

-- Domains are pointed & Least fixed points exist
bottom :: a
bottom = undefined
fix :: (a -> a) -> a
fix f = f (fix f)

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

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

-- Semantic "seq"
semseq :: V -> V -> V
semseq x y = case x of
                 (Z _)        -> y ;
                 (FV _)       -> y ;
                 (Tagged _ _) -> y

-- Alternation
fatbar :: (a -> Maybe b) -> (a -> Maybe b) -> (a -> Maybe b)
f `fatbar` g = \ x -> (f x) `fb` (g x)
        where fb :: Maybe a -> Maybe a -> Maybe a
              Nothing `fb` y = y
              (Just v) `fb` y = (Just v)
```

Fig. 3. Semantic Operators

The last three operations, Kleisli composition (`<>`), alternation (`fatbar`), and purification (`purify`) are all integral to modeling Haskell constructs involving fine control of demand such as `case` expressions, patterns, guards, and multi-line declarations. The Kleisli composition (`<>`) is used as the control operator for pattern matching. Given `(f :: a -> Maybe b)` and `(g :: b -> Maybe c)`, `(f <> g)` is the function which, applied to an input `x`, performs `(f x)` first. If `(f x)` produces `(Just v)`, then the value of `((f <> g) x)` is `(g v)`. Otherwise if `(f x)` produces `Nothing`, then `((f <> g) x)` is `Nothing`. If `f` is the meaning of a pattern, then this has similar behavior to pattern matching, because the failure of the match (i.e., signified by `Nothing`) is propagated. This will be expanded upon in Section 5.2.

The `fatbar` operation[2] is integral to the specification of `case` expressions. In `((fatbar m1 m2) v)`, if `(m1 v)` is `Nothing` (indicating a pattern match failure), then the result is the same as `(m2 v)`; otherwise, `(m1 v)` is returned. This sequencing behavior is very close to the meaning of the Haskell case expression `(case v of { m1 ; m2 })`.

In the semantics, purification distinguishes the computational aspects of the language from its pure, value aspects, and the operator `purify` signifies a return from the computational world to the value world. While fine control of demand may occur within Haskell expressions based upon pattern-matching, the meaning of such expressions will be a value in `V` rather than a computation in `(Maybe V)`, and the "run" of the `Maybe` monad, `purify`, converts a computation into a value. Note that a match failure (`Nothing`) is recast as a `bottom` by `purify`, and this reflects unrecoverable errors such as exhausting all the branches of a `case`.

4 The Meaning of Patterns

Pattern matching of nested patterns is a challenging problem when describing the semantics of any functional programming language, and we consider our treatment of nested patterns to be one of the major contributions of this paper. In Haskell, patterns can occur within several different syntactic contexts: lambda expressions (`\ p1 p2 -> e`), let expressions (`let p = e1 in e2`), matches (`case e1 of p -> e2`), and clauses in multi-line function definitions (`f p1 p2 = e`). Patterns may also appear as sub-patterns within other patterns—these are the so-called *nested patterns*.

With one important exception, Haskell patterns behave very much like patterns found in other functional languages such as ML[9], Clean[17], and Miranda[11]. Haskell contains a pattern operator (represented with a tilde `~`) which can be used to alter the default order of evaluation in a pattern match. Patterns of the form `~p` are known in the Haskell literature[13] as *irrefutable* patterns, although we shall see that this term is something of a misnomer because matches against irrefutable patterns can indeed fail. Haskell patterns without `~` are evaluated strictly (as in ML and Miranda, etc.). That is, matching a `~`-free pattern `p`

[2] The name comes from Peyton-Jones[11] where this same function was represented by a fat bar ‖—hence the name "fatbar."

```
data Tree = T Tree Tree | S Tree | R Tree | L
ex0 = (\  (T (S x)  (R y)) -> L) (T L (R L))      --->   match failure
ex1 = (\ ~(T (S x)  (R y)) -> L) (T L (R L))      --->   L
ex2 = (\ ~(T (S x)  (R y)) -> x) (T L (R L))      --->   match failure
ex3 = (\ ~(T ~(S x) (R y)) -> y) (T L (R L))      --->   L
ex4 = (\ ~(T (S x) ~(R y)) -> y) (T L (R L))      --->   match failure
```

Fig. 4. Shifting matching from binding to evaluation with ~

against a value v performs the entire pattern match at the binding-time of variables within p. By contrast in a pattern-match of ~p against v, any matching of p is delayed until a variable within p is evaluated. In that sense, the pattern ~p is lazier than p. The ~ operator in Haskell splits the evaluation of a pattern match between the binding and evaluation times of pattern variables.

Several examples will help clarify how the pattern operator ~ splits up the evaluation of a pattern match. The Haskell function applications (ex0-ex4) from Figure 4 demonstrate some of the evaluation subtleties arising from the use of irrefutable patterns. They all involve variations on the pattern (T (S x) (R y)) applied to a value (T L (R L)) (call it v). Distinguish each pattern in ex0-ex4 as p affixed with the expression number (e.g., p1 is "~(T (S x) (R y))"). Note also that the lambda expressions of ex0 and ex1 have the constant body L, while those of ex2-ex4 have the variable bodies x and y. Patterns p1-p4 include ~, and this, in combination with the different bodies results in the (different) evaluation behavior as is noted on the right-hand side of the figure.

Because it contains no ~, the pattern match of p0 against v in ex0 is performed entirely at the binding time of variables x and y, which results in a pattern match failure, although neither x nor y are evaluated in the body of ex0. ex1 is identical to ex0 except that p1 has a ~ affixed. Evaluating ex1 succeeds because the irrefutable pattern p1 has shifted the entire pattern match to the evaluation times of its variables x and y, and because neither variable is evaluated in the body of ex1, the failure-producing pattern match is never performed. However, if we were to evaluate one of these variables (as in ex2), a match failure would be produced as before with ex0. In ex3, the evaluation of y in the body forces the match of part of p3 under the outermost ~ (i.e., in "~(T ...)"), but the second ~ (in "~(S x)") allows the whole match to succeed. The variable y is evaluated in the body of ex4 and causes a match failure, despite the fact that the subpattern ~(R y) matches the (R L) in argument v. Interestingly in this case, evaluating y forces the match of (S x) against L as in previous failure cases.

4.1 How do we model patterns?

A pattern p may be modeled as a function taking a value (the object being matched against) which produces either bindings for the variables in p or an error signifying match failure. We will refer to the variables of a pattern as its *fringe*. Specifying potentially error-producing functions is usually accomplished with an error monad[8]; in Haskell, we use the built-in Maybe monad for this

purpose. We view a pattern `p` as a function which takes a value and returns a tuple of values containing bindings for the fringe of `p`. Because we model arbitrary tuples as lists, the type of the semantic function `mP` is `P -> V -> Maybe [V]`. If `(mP p v)` is `(Just vs)`, then the list of values `vs` are the bindings for the fringe of `p`. Pattern match failure occurs if `(mP p v)` returns `Nothing`.

We define a special function `fringe` for calculating the fringe of a pattern, such that `(fringe p)` returns the fringe of `p` in order of occurrence from left to right:

```
fringe :: P -> [Name]        fringe (Pcondata n ps) =
fringe (Pconst i) = []                    foldr (++) [] (map fringe ps)
fringe (Pvar x)   = [x]      fringe (Ptuple ps) =
fringe Pwildcard  = []                    foldr (++) [] (map fringe ps)
fringe (Ptilde p)  = fringe p
```

Matching a variable never forces any further matching and always succeeds, so the variable `x` should be bound to the value `v`. This is signified by returning `(Just [v])`. To match a pattern constant, the argument `i` must be compared against the constant `k` in the pattern. If not equal, return `Nothing`, signifying pattern match failure. Otherwise, return `(Just [])`. Observe that because no additional bindings were found, that the empty list is returned. The simplest pattern is the wildcard pattern, which always succeeds, returning `(Just [])`.

```
mP  :: P -> V -> Maybe [V]
mP (Pvar x) v                          = Just [v]
mP (Pconst k) (Z i)                    = if i==k then Just [] else Nothing
mP Pwildcard v                         = Just []
mP (Ptuple ps) (Tagged "tuple" vs)     = stuple (map mP ps) vs
mP (Pcondata n ps) (Tagged t vs)       = if n==t then
                                               stuple (map mP ps) vs
                                         else Nothing
mP (Pnewdata n p) v                    = mP p v
mP (Ptilde p) v                        = Just(purifyN (arity p) (mP p v))
   where purifyN n x = project 0 (map purify (replicate n x))
         project i (x:xs) = (x !! i) : (project (i+1) xs)
         project i []     = []
         arity = length . fringe

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')
replicate 0 x         = []
replicate n x         = x : (replicate (n-1) x)
```

Fig. 5. Semantics of patterns

Tuples and structured data have much in common so we discuss both these cases together. They both use the auxiliary function `stuple` which performs sufficient evaluation to match against the pattern, and no more. Matching a Haskell pattern `(p1,...,pn)` against a tuple value `(v1,...,vn)` succeeds only when each match of `pi` against `vi` succeeds. In that sense, matching tuple patterns in Haskell is somewhat strict. The function `stuple` (for *strict tuple*) takes a list `qs` of pattern meanings (i.e., functions of type `(V -> Maybe [V])`) and a corresponding list of values `vs`, applies each pattern meaning `qi` to the corresponding `vi`. If all of these matches succeed, then the bindings collected from each match are returned. However, if one of the matches fails, then `Nothing` is returned.

Structured data is similar. When pattern `(Pcondata n ps)` is matched against `(Tagged t vs)`, first the tags `n` and `t` are compared. If the tags are equal, then `ps` are matched against `vs` as in the `Ptuple` case. Otherwise, `Nothing` is returned.

The constructor function introduced by a `newtype` declaration acts like the identity function. Thus to match a `Pnewdata` pattern against an argument, just match its sub-pattern against the argument. A similar rule is found in the evaluation of `newtype` constructor functions in the function `mE`.

The most semantically interesting pattern is the irrefutable pattern `~p`, whose semantics is shown in Figure 5. Laziness of the match of `(Ptilde p)` against `v` is achieved by wrapping any actual matching by the `Just` constructor.

$$\texttt{purifyN n (mP p v)} = \begin{cases} [\texttt{v}_1, \ldots, \texttt{v}_\texttt{n}], & \text{if } \texttt{(mP p v)} = \texttt{Just}[\texttt{v}_1, \ldots, \texttt{v}_\texttt{n}] \\ [\texttt{bottom}, \ldots, \texttt{bottom}], & \text{if } \texttt{(mP p v)} = \texttt{Nothing} \end{cases}$$

where `n` is `(arity p)`. Notice that this is almost identical to `(purify (mP p v))`, except that the `Nothing` branch returns a list of `bottom` values—one for every variable in `(fringe p)`. Generally, the effect of deferring pattern match failure is characterized by the following equivalence:

$$\texttt{(mP ~p v)} = \texttt{Just[bottom,...,bottom]} \iff \texttt{(mP p v)} = \texttt{Nothing}$$

Given this view of patterns, it is now possible to describe formally how the two varieties of pattern match failure—binding and evaluation time failure—are manifested in the semantics. The binding time failure for a match `(mP p v)` of pattern `p` against value `v` is manifested by a `Nothing` *computation*. If Haskell did not include irrefutable patterns, that would be the whole story. If the match `(mP p v)` is `Nothing`, then the match `(mP ~p v)` will return `(Just [bottom,...,bottom])`, and each variable in the fringe of `~p` will be bound to `bottom`. This is precisely how the pattern match is deferred until evaluation time. Instead of failing at binding time, each variable in `(fringe p)` is bound to `bottom`, and so the match failure will be reported at the evaluation times of `(fringe p)` (if at all).

5 The Meaning of Expressions

In our semantics, the meaning of a Haskell program is an environment to value function. This section contains a detailed explanation of the semantic function

for expressions `mE`. Each of the following subsections describes a few related clauses making up the definition of `mE` along with the auxiliary functions necessary for those clauses. These auxiliary functions supply meaning for the other syntactic categories of Haskell. The semantics of Haskell's mutually recursive `let` expressions is deferred until the discussion of declarations in Section 6.1.

```
mE :: E -> Env -> V
mE (Var n) rho         = rho n
mE (Const i) rho       = (Z i)
mE (TupleExp es) rho   =
 tuple $ map (\e-> mE e rho) es
mE (Cond e0 e1 e2) rho =
 ifV (mE e0 rho) (mE e1 rho) (mE e2 rho)
mE Undefined rho       = bottom
```

```
ifV :: V -> a -> a -> a
ifV (Tagged "True" []) x y = x
ifV (Tagged "False" []) x y = y

tuple :: [V] -> V
tuple [v] = v
tuple vs = Tagged "tuple" vs
```

Fig. 6. Semantics of simple expressions

For pedagogical reasons, the text of the paper breaks the definitions into a few clauses at a time. An actual implementation would need to collect them all together in a single place. Because this paper is intended for a general audience, constructs which are unique to Haskell are described operationally first when appropriate.

5.1 Simple Expressions

The meaning of a variable is obtained by extracting its binding from the current environment (`rho`) by applying it to the variables name. Constants are simply turned into values. In the case for tuples, the laziness of Haskell first becomes apparent. The evaluation of each of the subexpressions (the `es`) must be suspended, and this is accomplished with the lazy list constructor `[]`. Conditional expressions, are easily translated using the control operator `ifV`. The meaning of the `Undefined` expression is simply `bottom`.

5.2 Application and Abstraction

Function application and abstraction are the essence of any functional language. Because nested patterns are λ-bound in Haskell (as in most functional languages), care must be taken to implement Haskell's laziness correctly.

To compute the meaning of an application `App`, use `app` to apply the meaning of `e1` to the meaning of `e2`. The meaning of Haskell abstraction is defined in terms of an auxiliary operation called `lam`, which we describe in detail below. The meaning of an abstraction with a single pattern `(Abs [p] e)` in environment `rho` is simply `(lam p e rho)`, injected into `V` by `FV`. Haskell abstractions may have more than one pattern bound in a lambda; in the abstract syntax, this is: `(Abs`

```
mE :: E -> Env -> V
mE (App e1 e2) rho = app (mE e1 rho) (mE e2 rho)
mE (Abs [p] e) rho = FV $ lam p e rho
mE (Abs ps e) rho  = sharp (length ps) [] (lam (ptuple ps) e rho)

lam :: P -> E -> Env -> V -> V
lam p e rho =
        (mP p <> ((\vs -> mE e (extend rho xs vs)) >>> Just)) >>> purify
             where xs = fringe p
ptuple :: [P] -> P
ptuple [p] = p
ptuple ps = Pcondata "tuple" ps
```

Fig. 7. Semantics of application and abstraction

`[p1,...,pn] e`). We note that this construct is *not* reducible to the previous case as: `(Abs [p1] (... Abs [pn] e))`. It is, in fact, lazier than this translation (cf. [13], section 3.3), because `(Abs [p1,...,pn] e)` waits for all `n` arguments before matching any of the patterns, while `(Abs [p1] (... Abs [pn] e))` matches the arguments as each is applied. Laziness for `(Abs [p1,...,pn] e)` is achieved by currying (with `sharp`) `n` times.

What of `lam`? Evaluating the Haskell expression (`\p-> e`) applied to value `v` in environment `rho` follows this sequence of steps. First, match `p` against `v`. Secondly, if this is `(Just vs)`, then evaluate `e` in the extended environment `(extend rho xs vs)` where `xs` is the fringe of `p`; if the match produces `Nothing`, then the whole application should fail. As observed in Section 3.2, this kind of sequencing suggests using Kleisli composition (`<>`). These two steps can be neatly characterized as:

```
(mP p <> ((\vs -> mE e (extend rho xs vs)) >>> Just)) :: V -> Maybe V
```

where `xs` is the fringe of `p`. Because function values are functions from `V` to `V`, we define `(lam p e rho)` as the above expression composed on the right by `purify`:

```
(mP p <> ((\vs -> mE e (extend rho xs vs)) >>> Just)) >>> purify  :: V->V
```

This is an example of how purification delimits the computational aspects of Haskell. Although an abstraction `(\p -> e)` has effects arising from pattern matching, these impurities are eliminated by post-composing with `purify`.

5.3 The Meaning of Guarded Expressions

A body is a form of guarded Haskell expression occurring only within the scope of a `case` expression, function declaration, or pattern binding. For example in the following `case` expression:

```
case e of { p | g1 -> e1 ; ... | gn -> en where { decls } ; <rest> }
```

the "`| g1 -> e1 ;...; | gn -> en`" is a body, where the guards `g1`, ..., `gn` are boolean-valued expressions. This body would be represented in the abstract syntax as `(Guarded [(g1,e1),...,(gn,en)])`. A body may also be unguarded (i.e., represented as `(Normal e)` where `e` is just an expression in `E`).

Operationally, bodies with guards behave like nested *if-then-else* expressions. The body "`| g1 -> e1 ;...; | gn -> en`" within the aforementioned `case` expression would evaluate as follows:

1. If the pattern match of `p` against the value of `e` fails, then continue with `<rest>`.
2. Otherwise if the pattern match succeeds, evaluate `g1`, ..., `gn` in ascending order until either a true guard `gi` is found, or the list is exhausted. If `gi` is the first true guard, then continue evaluating `ei`. If all the guards are false, then continue with `<rest>`.

Guarded bodies restrict when branches of a `case` expressions are taken. In particular, two identities hold:

```
(1) case v of { p | True -> e ; ... }  = case v of { p -> e ; ... }
(2) case v of { p | False -> e ; ... } = case v of { ... }
```

The first identity shows that having a constant `True` guard is identical to having no guard at all, while in case `(2)`, having a constant `False` guard on the first branch is equivalent to always ignoring the branch altogether.

```
mB :: B -> Env -> Maybe V
mB (Normal e) rho         = Just (mE e rho)
mB (Guarded gl) rho       = ite gl rho
  where ite [] rho         = Nothing
        ite ((g,e):gs) rho = ifV (mE g rho) (Just (mE e rho)) (ite gs rho)
```

Fig. 8. Semantics of guarded expressions `B` (bodies)

Because bodies occur within the branches of `case` statements, they are `Maybe`-valued. Figure 8 displays the semantics of bodies, `mB :: B -> Env -> Maybe V`. The meaning of an unguarded body `(Normal e)` is just the meaning of the expression `e` injected into the `Maybe` monad. The meaning of a body `(Guarded [(g1,e1),...,(gn,en)])` is:

```
ifV (mE g1 rho) (Just (mE e1 rho))
                    ...
    (ifV (mE gn rho) (Just (mE en rho)) Nothing)
```

which behaves as a sequence of nested *if-then-else* where the last "else" clause is a `Nothing`. In this context, the `Nothing` signifies that the falsity of the guards has forced an otherwise successful branch in a `case` to fail, and that the next branch in the `case` should be attempted.

```
mE :: E -> Env -> V
mE (Case e ml) rho = mcase rho ml (mE e rho)

mcase :: Env -> [Match] -> V -> V
mcase rho ml = (fatbarL $ map (match rho) ml) >>> purify
      where fatbarL :: [V -> Maybe V] -> V -> Maybe V
            fatbarL ms = foldr fatbar (\ _ -> Nothing) ms

-- The function match is used to construct the meaning of a Match
match :: Env -> (P, B, [D]) -> V -> Maybe V
match rho (p,b,ds) = mP p <> (\vs -> mwhere (extend rho xs vs) b ds)
      where xs = fringe p

-- (mwhere rho b ds) is the meaning of body b in where-clause "b where ds"
mwhere ::  Env -> B -> [D] -> Maybe V
```

Fig. 9. Case expressions. Note that `mwhere` is defined in Figure 12.

5.4 Case Expressions

We next consider the semantics of `case` expressions. All of the definitions discussed in this section are summarized in Figure 9. The meaning of a `case` expression is defined in terms of two main auxiliary functions, `match` and `mcase`. The function `match` gives the meaning of a `Match`, which is really just a branch in a `case` expression. The function `mcase` uses the `fatbar` semantic control operator to construct the meaning of a `case` statement.

A match is a tuple, (`(p,b,ds) :: Match`), whose semantic specification is similar to that of `lam` (discussed in Section 5.2), although because it occurs within a `case` expression, it is `Maybe`-valued. An additional consideration is that the declarations in `ds` are visible within the body `b`. Such declarations would appear in Haskell concrete syntax as a "`where`" clause (i.e., "`b where ds`"). We have a third auxiliary function, `mwhere`, which models the mutually recursive `where` clauses of Haskell, but we defer discussion of it until Section 6.1 where such declarations are handled. It suffices to say that `(mwhere rho b ds :: Maybe V)` is the meaning of body `b` within the scope of `rho` extended to contain the bindings from `ds`. And, if `ds` is empty, then `(mwhere rho b ds)` is simply `(mB b rho)`.

Function `(match rho p b ds)` is defined as:

```
mP p <> (\vs -> mwhere (extend rho xs vs) b ds)
```

where `xs` is the fringe of `p`. We see the same pattern as in `lam`, where the bindings for the fringe of `p` are extracted by the Kleisli composition: `(mP p <> (\vs -> ...))`. The extended environment is then passed on to a call to `(mwhere (extend rho xs vs) b ds)`.

The function `mcase` takes the current environment, `rho`, and a list of `Match`s, `[m1,...,mn]`, and returns a function from `V` to `V`. Unfolding the definitions of

`fatbarL` in the definition of `mcase`, the value of `(mcase rho [m1,...,mn])` can be seen to have the form:

```
( (match rho m1) `fatbar`
                  ...
            (match rho mn) `fatbar` (\ _ -> Nothing))  ) >>> purify
```

Here we use infix notation `` `fatbar` ``.

Recall from Section 3.2 that, when the above term is applied to a value `x`, the control operator `fatbar` will sequence through the `((match rho mi) x)` from left to right, until coming to the leftmost computation of the form `(Just v)` (if it exists). The value of the above expression would be in that case `(purify (Just v))` or simply `v`. If all the `((match rho mi) x)` are `Nothing`, then the value of the above term is `(purify Nothing)` or simply `bottom`. This last eventuality occurs when all of the branches of a `case` expression have been exhausted.

Given the function `mcase`, it is a simple matter to define `case` expressions as in Figure 9. This is another example of how purification delimits the computational aspects of Haskell. Failure in the computational world (i.e., a `Nothing`) resulting from the interaction of patterns, cases, and guards is transmuted into value-level failure (i.e., `bottom`) by post-composing with `purify`.

5.5 Constructor Application

Constructor applications are evaluated in a manner much like tuples. The key difference is the possibility that constructors can have some of their arguments annotated as strict arguments. For example in Haskell, we might write `(data T = C String !Int)`, where the `!` signifies that the evaluation of second argument to `C` should be strict. We represent this in our abstract syntax by annotating each sub-argument to a constructor application `ConApp` with a strictness annotation of type `LS`.

To force evaluation of a strict constructor argument, we make use of the `semseq` semantic operator defined in Figure 3. If expression `e` is an argument to a strictly-annotated constructor, then the correct value for `e` in the resulting `Tagged` value should be `(semseq (mE e rho) (mE e rho))` for current environment `rho`. This may seem odd at first, but this expression is not identical to `(mE e rho)`, because they have different termination behavior. That is, `semseq` will force evaluation of `(mE e rho)`, and this may fail, causing the entire `(semseq (mE e rho) (mE e rho))` to fail.

Sequencing through the arguments of constructor application `(ConApp n el)` is performed with the auxiliary function `evalL`. `evalL` tests each strictly-annotated argument in `el` with `semseq` as outlined above, and then constructs and returns the corresponding `Tagged` value.

Below is a sample Hugs session showing the application of `mE` to two constructor applications, which we represent for readability in Haskell's concrete syntax. Both are applications of constructors `L` and `S` to the `Undefined` expression, but it is assumed that the `S` constructor has been declared with a strictness annotation (i.e., with "!"). Evaluating `(mE (L Undefined) rho0)` (for

```
mE :: E -> Env -> V
-- Strict and Lazy Constructor Applications
mE (ConApp n el) rho  = evalL el 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])

-- New type constructor applications
mE (NewApp n e) rho   = mE e rho

-- Miscellaneous Functions
mE (Seq e1 e2) rho    = semseq (mE e1 rho) (mE e2 rho)
mE (Bin op e1 e2) rho = binOp op (mE e1 rho) (mE e2 rho)
      where binOp Plus (Z i) (Z j)    = Z $ i+j
            binOp Mult (Z i) (Z j)    = Z $ i*j
            binOp IntEq (Z i) (Z j)   = Tagged (equal i j) []
            binOp IntLess (Z i) (Z j) = Tagged (less i j) []
            equal i j = if i==j then "True" else "False"
            less i j  = if i<j then "True" else "False"
```

Fig. 10. Semantics of constructor applications, `seq`, and arithmetic operations

some environment `rho0`) just results in the `Tagged` value `(L ?)` being returned (in pretty-printed form) as one would expect of a lazy constructor. Evaluating `(mE Undefined rho0)` produces a failure, because `semseq` forces the evaluation of the argument `Undefined`.

```
Semantics> mE (L Undefined) rho0
        (L ?)
Semantics> mE (S Undefined) rho0
        Program error: {undefined}
```

5.6 Newtype Constructor Application

A `newtype` constructor acts like the identity function, thus it is easy to define the clause of `mE` for `newtype` constructors.

5.7 Miscellaneous expressions

Finally, we come to the last few miscellaneous expression forms for the `Seq` operation and the primitive binary operators. We assume that the primitive operators are strict in their operands.

```
mD :: D -> Env -> V
mD (Fun f cs) rho  = sharp k [] body
      where
           body = mcase rho (map (\(ps,b,ds) -> (ptuple ps, b,ds)) cs)
           k  = length ((\(pl,_,_)->pl) (head cs))

mD (Val p b ds) rho = purify (mwhere rho b ds)
```

Fig. 11. Semantics of declarations `D`

6 The Meaning of Declarations

In this section, we consider how declarations are processed within Haskell. There are two declaration forms, function and pattern, represented in the abstract syntax as `(Fun f cs)` and `(Val p b ds)`, respectively. Here, `cs` is a list of `Clauses`; that is, `(cs :: [([P],B,[D])])`. The semantics for declarations `D`, discussed below, is presented in Figure 11.

The `Fun` declaration corresponds to a multi-line function declaration, such as:

```
nth :: Int -> [a] -> a
nth 0 (x:xs) = x
nth i (x:xs) = nth (i-1) xs
```

Observe that each clause necessarily has the same number of pattern arguments from the assumed well-typedness of terms.

The Haskell98 report[cf. Section 4.4.3, page 54] defines multi-line function declarations through translation into a `case` expression

> *The general binding form for functions is semantically equivalent to the equation (i.e. simple pattern binding):*
>
> $$x = \backslash x_1 \ldots x_k \texttt{->} \texttt{case}\ (x_1, \ldots, x_k)\ \texttt{of}\ \begin{array}{l}(p_{11}, \ldots, p_{1k})\ \ match_1 \\ \vdots \\ (p_{m1}, \ldots, p_{mk})\ \ match_m\end{array}$$
>
> *where the "x_i" are new identifiers.*

Let us now consider how to model `(Fun f cs)` in environment `rho`. We reuse `mcase` here to model the `case` expression behavior, and following the Haskell98 report quoted above, we make each `Clause` in the `cs` into a branch of the case:

```
mcase rho (map (\(ps,b,ds) -> (ptuple ps, b,ds)) cs) :: V -> V
```

This term is a function from `V` to `V`, but we can say a little more about the `V` argument. In general, it will expect a tuple value `(Tagged "tuple" vs)` as input because each branch of the case is made into a tuple pattern by `ptuple` (defined in Figure 7). But `f` is a function in curried form, whose arity, `k`, is the number of arguments to `f`. So we must apply currying `k` times using the `sharp`

semantic operator. The full definition of `(mD (Fun f cs))` appears in Figure 11. Interestingly, using the currying operator `sharp` dispenses with the condition "*where the "x_i" are new identifiers*" from the quote above by using variables which are fresh in the metalanguage.

The Haskell98 report[cf. Section 4.4.3, page 54] defines pattern declarations through translation into a `let` expression. It begins by giving the general form of pattern bindings:

> *[I]n other words, a pattern binding is:*
>
> $$\begin{array}{l} p \mid g_1 = e_1 \\ \quad \mid g_2 = e_2 \\ \quad \ldots \\ \quad \mid g_m = e_m \\ \quad \texttt{where}\ \{decls\} \end{array}$$
>
> **Translation:** *The pattern binding above is semantically equivalent to this simple pattern binding:*
>
> $$\begin{array}{l} p = \texttt{let}\ decls\ \texttt{in} \\ \quad \texttt{if}\ g_1\ \texttt{then}\ e_1\ \texttt{else} \\ \quad \texttt{if}\ g_2\ \texttt{then}\ e_2\ \texttt{else} \\ \quad \ldots \\ \quad \texttt{if}\ g_m\ \texttt{then}\ e_m\ \texttt{else error}\ \text{"}Unmatched\ pattern\text{"} \end{array}$$

This translation means that a pattern binding may be reduced to a related `let` expression. In Section 5.4, we made use of the function `(mwhere :: Env -> B -> [D] -> Maybe V)` which models Haskell's `where` clauses, and we make use of that function again here to specify the let-binding `(let decls in...)` above as: `mD (Val p b ds) rho = purify (mwhere rho b ds)`. Observe that the "`(mwhere rho b ds :: Maybe V)`" is a computation and must be brought into the value world using `purify`.

6.1 Mutual Recursion and Let-binding

Our technique for modeling mutually recursive declarations in Haskell adapts a standard technique from denotational semantics for specifying mutual recursion and recursive let expressions. However, this technique applies only to the lazy λ-calculus in which only variable and tuple patterns are λ-bound, and so care must be taken when generalizing it to Haskell (where nested patterns are also λ-bound). In this section, we overview the standard technique, compare it with the definition in the Haskell98 report, and describe our specification of mutual recursion in Haskell.

To overview how mutual recursion is typically specified denotationally, we consider adding various let-binding constructs to the lazy λ-calculus. Let us say that we have a semantics for the non-recursive lazy λ-calculus, $[\![-]\!] : \mathit{Lazy} \to \mathit{env} \to \mathit{Value}$, where *env* and *Value* are defined similarly to `Env` and `V`, respectively.

Non-recursive let expressions are frequently introduced as syntactic sugar for an application:

$$[\![\text{let } x = e \text{ in } e']\!] =_{def} [\![(\lambda x.e')e]\!] \tag{1}$$

The non-recursive let-binding of variable x to e in e' is accomplished merely by function application (which is already handled by $[\![-]\!]$). Handling recursive let-binding follows a similar pattern, although in this case, an explicit use of the least fix point operator fix becomes necessary:

$$[\![\text{letrec } x = e \text{ in } e']\!]\rho =_{def} [\![(\lambda x.e')]\!]\rho\ (fix([\![\lambda x.e]\!]\rho)) \tag{2}$$

Because e may contain references to the recursively defined x, one must apply fix to resolve the recursion.

This last definition handles only one recursive binding ($x = e$). There is a standard technique in denotational semantics for extending Equation 2 to sets of mutually recursive bindings using tuples. In the case of mutually recursive bindings, we are given a set of mutually recursive bindings, $\{x_1 = e_1, \ldots, x_n = e_n\}$, that we refactor into a single tuple pattern $(x_1, \ldots, x_n)$ and tuple expression $(e_1, \ldots, e_n)$. Now, this pattern and expression play the same rôle as x and e did in Equation 2:

$$\begin{aligned}[\![\text{letrec } \{x_1 = e_1 \ldots x_n = e_n\} \text{ in } e']\!] =_{def} \\ [\![(\lambda\langle x_1, \ldots, x_n\rangle.e')]\!]\rho\ (fix([\![\lambda\langle x_1, \ldots, x_n\rangle.\langle e_1, \ldots, e_n\rangle]\!]\rho))\end{aligned} \tag{3}$$

Now returning to mutually recursive bindings in Haskell, something very similar to the standard technique occurs. The only complications arise in that, in Haskell, nested patterns are λ-bound and not just variables or tuples. Comparing Equation 3 to the relevant equations from the Haskell98 report[cf. Section 3.12, page 22], we can see that something very similar is going on:

(a) $\texttt{let}\ \{\ p_1 = e_1\ ; \ldots ;\ p_n = e_n\ \}\ \texttt{in}\ e_0 =$
$\qquad\qquad \texttt{let}\ (\tilde{}p_1, \ldots, \tilde{}p_n) = (e_1, \ldots, e_n)\ \texttt{in}\ e_0$

(b) $\texttt{let}\ p = e_1\ \texttt{in}\ e_0 = \texttt{case}\ e_1\ \texttt{of}\ \{\ \tilde{}p \texttt{->} e_0\ \}$
where no variable in p appears free in e_0.

(c) $\texttt{let}\ p = e_1\ \texttt{in}\ e_0 = \texttt{let}\ p = \texttt{fix}\ (\backslash\tilde{}p \texttt{->} e_1)\ \texttt{in}\ e_0$

Definition (b) shows how to convert a simple `let` into a `case` expression in a manner similar to that of Equation 1. Definitions (a) refactors mutually recursive bindings into a single tuple pattern and tuple expression, and (c) resolves the recursion with an explicit fix point. It is worth pointing out that the use of `fix` in (c) is really hypothetical and is meant to direct the reader to the implicit intentions of Haskell's designers; Haskell does not contain a `fix` operator and one must define it as we have in Figure 3.

One semantic subtlety in (a)-(c) arises from the fact that pattern matching perturbs Haskell's default lazy evaluation. A Haskell abstraction `(\p -> e)` may

```
mE :: E -> Env -> V
mE (Let ds e) rho     = letbind rho ds e

letbind ::  Env -> [D] -> E -> V
letbind rho [] e = mE e rho
letbind rho ds e = (lam dp e rho) v
 where
  dp = tildefy (declared ds)
  xs = frD ds
  decls env = tuple (map (\d -> mD d env) ds)
  v = fix  (((mP dp) <>
              ((\vs -> decls (extend rho xs vs)) >>> Just)) >>> purify)

mwhere ::  Env -> B -> [D] -> Maybe V
mwhere rho b [] = mB b rho
mwhere rho b ds = (wherecls dp b rho) v
 where
  dp = tildefy (declared ds)
  xs = frD ds
  decls env = tuple (map (\d -> mD d env) ds)
  v = fix (((mP dp) <>
             ((\vs -> decls (extend rho xs vs)) >>> Just)) >>> purify)
  wherecls p b rho = (mP p <> (\vs -> mB b (extend rho xs vs)))
                        where xs = fringe p

-- the fringe of a declaration D
frD :: [D] -> [Name]
frD [] = []
frD ((Fun f _):ds) = f : (frD ds)
frD ((Val p _ _):ds) = fringe p ++ (frD ds)
```

Fig. 12. Semantics of mutually recursive bindings

be partially strict in that an argument to the abstraction will be evaluated against `p` to get the bindings for `(fringe p)`. Because arguments to `fix` must be lazy, care must be taken to annotate certain patterns with the irrefutable pattern operator `~`, and this is why `~` pops up somewhat mysteriously in definitions (*a*)-(*c*). Our specification of mutual recursion will make similar `~` annotations where necessary. We condense (*a*)-(*c*) into the following schemas, which guides our specification of mutually recursive let-binding in Haskell:

$$\begin{aligned}&\texttt{let}\ \{\ p_1 = e_1\ ;\ \ldots\ ;\ p_n = e_n\ \}\ \texttt{in}\ e\ = \\ &\qquad (\backslash\ \tilde{}(\tilde{}p_1, \ldots, \tilde{}p_n)\ \texttt{->}\ e)\ (\texttt{fix}\ (\backslash\ \tilde{}(\tilde{}p_1, \ldots, \tilde{}p_n)\ \texttt{->}\ (e_1, \ldots, e_n)))\end{aligned} \tag{4}$$

To be concrete, let us consider what must be done to specify `(Let ds e)` in the manner discussed above. First, we must gather all of the patterns in the left-hand sides of the declarations in `ds` (call them `p1`, ..., `pn`) and form the necessary tuple pattern: `~(~p1,...,~pn)`. This is accomplished chiefly with two auxiliary functions, `tildefy` and `declared` (both shown below). `tildefy` adds a `~` to a pattern if necessary. Note that a variable pattern `(Pvar x)` needs no `~` and a no redundant `~`s need be added, either. `(declared ds)` returns a tuple pattern in which all component patterns have been `tildefy`'d.

```
declared :: [D] -> P
declared ds = ptuple $ map getbinder ds
    where  getbinder (Fun f _)   = Pvar f
           getbinder (Val p _ _) = tildefy p

tildefy :: P -> P
tildefy p = case p of  (Ptilde p') -> p
                       (Pvar x)    -> p
                       _           -> (Ptilde p)
```

The next step in specifying `(Let ds e)` is to form a tuple value out of the right-hand sides of its declarations `ds`. This corresponds to the $(e_1, \ldots, e_n)$ tuple in (*a*) above. This is accomplished mainly by mapping the semantics of declarations, `mD`, onto the declaration list `ds`, and then converting the list into a tuple value. Recall `tuple` is defined in Figure 6.

Now we can put all of these pieces together into the auxiliary function `letbind`. `(letbind rho ds e)` takes the current environment `rho`, extends it with the mutually recursive bindings from `ds`, and evaluates `e` in this extended environment. In other words, `(letbind rho ds e)` is precisely `(mE (Let ds e) rho)`. `(letbind rho ds e)` implements the scheme given in Equation 4 above, and we define the meaning of Haskell `let` with it in Figure 12.

Defined in an analogous manner to `letbind` is the auxiliary function `mwhere`. This has been used to describe `where` clauses around bodies in `B`. `letbind` and `mwhere` handle mutual recursive bindings identically, and the principal difference between them is that `mwhere` applies to bodies `B`, and hence has a computational type.

```
e1 = seq ((\ (Just x) y -> x) Nothing) 3
e2 = seq ((\ (Just x) -> (\ y -> x)) Nothing) 3
e3 = (\ ~(x, Just y) -> x) (0, Nothing)
e4 = case 1 of
          x | x==z -> (case 1 of w | False -> 33)
                where z = 1
          y -> 101
e5 = case 1 of
          x | x==z -> (case 1 of w | True -> 33)
                where z = 2
          y -> 101
e6 = let  fac 0 = 1
          fac n = n * (fac (n-1))
     in fac 3
```

```
Semantics> mE e1 rho0
    3
Semantics> mE e2 rho0
    Program error: {undefined}
Semantics> mE e3 rho0
    Program error: {undefined}
Semantics> mE e4 rho0
    Program error: {undefined}
Semantics> mE e5 rho0
    101
Semantics> mE e6 rho0
    6
Semantics>
```

```
Hugs> e1
    3
Hugs> e2
    Program error: {e2_v2550 Maybe_Nothing}
Hugs> e3
    Program error: {e3_v2558 (Num_fromInt instNum_v35 0,...)}
Hugs> e4
    Program error: {e4_v2562 (Num_fromInt instNum_v35 1)}
Hugs> e5
    101
Hugs> e6
    6
Hugs>
```

Fig. 13. Comparing the semantics to Hugs

7 Testing the Interpreter

Figure 13 presents a number of examples, and compares the output of the semantics (executing in Hugs) against that of the Hugs Haskell interpreter. In the figure, `rho0` is the empty environment, and, for the sake of readability, we have not shown the abstract syntax translations of `e1` through `e6`. Two interesting cases are `e1` and `e2`. As we observed in Section 5.2, the lambda expression `(\p1 p2 -> e)` is lazier than the explicitly curried expression `(\p1 -> \p2 -> e)`, and the semantics `mE` agrees with the Hugs interpreter on this point.

The semantics `mE` explains this somewhat surprising distinction nicely. Consider the following Haskell terms:

```
t1 = (\ (Just x) y -> x) Nothing
t2 = ((\ (Just x) -> (\ y -> x)) Nothing
```

`(mE t1 rho0)` is the function value `FV(`λ`_.bottom)`—that is, a function that, if applied, will fail. According to `mE`, the meaning of the application `t2` is `bottom`, because the pattern matching of `(Just x)` against `Nothing` is performed. `mE` also distinguishes between $\perp_{a \to b}$ and (λ`_.` $\perp$: $a \to b$) as required by the Haskell98 report[13] (cf. Section 6.2, page 75): `(semseq (mE t1 rho0) 3)` is simply `3`, while `(semseq (mE t2 rho0) 3)` is `bottom`.

8 Future Work and Conclusions

The Haskell98 report contains a number of translation schemas which describe the interactions between Haskell features, and by doing so, provide a semi-formal language definition to be used by programmers and language implementors alike. In Section 6.1, we included several such schemas to motivate our formal view of mutual recursion. These schemas may also be viewed as a set of axioms for Haskell which must be satisfied by any candidate semantics (including this one) for it to be considered a *bona fide* Haskell semantics in some sense. An example validation occurs below in Figure 14. We have validated a number of the translation schemas from in the Haskell98 report having to do with pattern-matching, but a number of schemas remain unchecked. A follow-on to this work would collect all of these Haskell "axioms" together with the proofs of their validation with respect to this semantics.

Haskell is commonly referred to as a lazy functional language, but it is more properly understood as a non-eager language because it contains features (patterns, the `seq` operator, etc.) which introduce strict perturbations of the default lazy evaluation mechanism. These perturbations are important for practical reasons: expert Haskell programmers may use strictness sparingly in their programs to avoid some of the computational overhead associated with laziness without giving it up entirely.

However, this mixed evaluation order complicates Haskell from a semantic point of view. We have modeled Haskell's control of demand by writing a calculational semantics in Haskell, relying on certain built-in aspects of Haskell (laziness, etc.) to model Haskell itself. An alternative would have modeled Haskell's

```
mE (case v of { _ -> e ; _ -> e' }) rho
    = mcase rho { _ -> e ; _ -> e' }) (mE v rho)
    = (((match rho (_) e) `fatbar`
                  (match rho (_) e')) >>> purify) (mE v rho)
    = ((((mP (_)) <> ((\ _ -> mE e rho) >>> Just) `fatbar`
                                ...)) >>> purify) (mE v rho)
          where ... = (mP (_)) <> ((\ _ -> mE e' rho) >>> Just)
    = ((((\_ -> Just []) <> ((\ _ -> mE e rho) >>> Just)
                       `fatbar` ...)) >>> purify) (mE v rho)
    = ((((\ _ -> mE e rho) >>> Just)
                           `fatbar` ...)) >>> purify) (mE v rho)
    = (((\ _ -> mE e rho) >>> Just) >>> purify) (mE v rho)
    = purify (Just (mE e rho))
    = mE e rho
```

Fig. 14. Validating the semantics w.r.t. translation "`case` v `of` { _–>e;_–>e' } = e"

fine control of demand by monadic interpreter[2, 8], which can model the full range, from fully strict to fully lazy languages.

The present work is the first formal treatment of the fine control of demand in Haskell, but we believe that many of the techniques presented here apply equally well to the semantics of functional languages in general. The patterns considered were nested patterns, and we did not resort to pattern-match compilation to simplify the task. The work clearly defines the interaction between `data` (with and without strictness annotations) and `newtype` data constructors with Haskell's other features.

The code presented in this paper is available online at: `www.cse.ogi.edu/~wlh`.

Acknowledgements

The authors would like to thank John Launchbury, Dick Kieburtz, and Mark Jones for their insights into Haskell and constructive criticism of earlier versions of this work. Both the anonymous referees and the Pacsoft research group at OGI offered many helpful suggestions that led to significant improvements in the presentation.

References

1. Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs.* McGraw Hill, Cambridge, Mass., second edition, 1996.
2. Olivier Danvy, Jürgen Koslowski, and Karoline Malmkjær. Compiling monads. Technical Report CIS-92-3, Kansas State University, Manhattan, Kansas, December 1991.

3. Daniel P. Friedman, Mitchell Wand, and Christopher T. Haynes. *Essentials of Programming Languages.* McGraw-Hill Book Co., New York, N.Y., second edition, 2001.
4. Murdoch Gabbay and Andrew Pitts. A new approach to abstract syntax involving binders. In G. Longo, editor, *Proceedings of the 14th Annual Symposium on Logic in Computer Science (LICS'99)*, pages 214–224, Trento, Italy, July 1999. IEEE Computer Society Press.
5. Carl A. Gunter. *Semantics of Programming Languages: Programming Techniques.* The MIT Press, Cambridge, Massachusetts, 1992.
6. Cordelia Hall, Kevin Hammond, Simon Peyton Jones, and Phillip Wadler. Type classes in haskell. In *Proceedings of the European Symposium on Programming*, volume 788 of *Lecture Notes in Computer Science*, pages 241–256. Springer Verlag, April 1994.
7. Mark P. Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. In *FPCA '93: Conference on Functional Programming and Computer Architecture, Copenhagen, Denmark*, pages 52–61, New York, N.Y., June 1993. ACM Press.
8. Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *Conference record of POPL '95, 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.*, pages 333–343. ACM Press, January 1995.
9. Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised).* The MIT Press, 1997.
10. Eugenio Moggi. Functor categories and two-level languages. In *Proceedings of the First International Conference on Foundations of Software Science and Computation Structure (FoSSaCS'98)*, volume 1378 of *Lecture Notes in Computer Science*, pages 211–223. Springer Verlag, 1998.
11. Simon Peyton Jones. *The Implementation of Functional Programming Languages.* Computer Science. Prentice-Hall, 1987.
12. Simon Peyton Jones. Implementing lazy functional languages on stock hardware: the spineless tagless G-machine. *Journal of Functional Programming*, 2(2):127–202, July 1992.
13. Simon Peyton Jones and John Hughes (editors). Report on the programming language Haskell 98. February 1999.
14. Simon Peyton Jones and Simon Marlowe. Secrets of the glasgow haskell compiler inliner. In *Proceedings of the Workshop on Implementing Declarative Languages (IDL'99)*, September 1999.
15. Simon Peyton Jones and André L. M. Santos. A transformation-based optimiser for Haskell. *Science of Computer Programming*, 32(1–3):3–47, September 1998.
16. Andrew Pitts and Murdoch Gabbay. A metalanguage for programming with bound names modulo renaming. In *Mathematics of Program Construction*, volume 1837 of *Lecture Notes in Computer Science*, pages 230–255. Springer Verlag, 2000.
17. 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.
18. Programatica Home Page. `www.cse.ogi.edu/PacSoft/projects/programatica`. James Hook, Principal Investigator.
19. Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Semantics.* MIT Press, Cambridge, Massachusetts, 1977.
20. Phillip Wadler. The essence of functional programming. *19th POPL*, pages 1–14, January 1992.

A Pattern-matching compilation is not just desugaring

If a pattern `p` is (`C` $t_1 \ldots t_n$) where t_i are variables, and `C` is a constructor function, then `p` is a *simple* pattern. However, if one or more of the t_i are not variable patterns, then `p` is a *nested* pattern. Pattern-matching compilation[11, 15] is typically performed as part of the front-end (as it is in GHC and Hugs), because it yields more efficient programs (see Chapter 5 by Wadler in [11] for further details). Figure 15 shows an example of pattern-match compilation in which the definition of a Haskell function `nodups` with nested patterns is transformed into a similar definition without nested patterns. One feature of this transformation was the necessity of generating new variables `x`, `x'`, `xs`, and `xs'` along the way.

```
nodups1 l =      -- Original with nested patterns:
  case l of
    [] -> []
    [x] -> [x]
    (y:(x:xs)) -> if x==y then (nodups1 (x:xs))
                          else (y:(nodups1 (x:xs)))

nodups2 xs'' =   -- After pattern-match compilation:
  case xs'' of
    [] -> []
     x':xs' -> case xs' of
                 [] -> [x']
                 x:xs -> if x'==x then (nodups2 (x:xs))
                                  else (x':(nodups2 (x:xs)))
```

Fig. 15. Pattern-match compilation is syntactic saccharin

Previous attempts[12, 15] to define a denotational semantics for the core of Haskell concentrate on the fragment of the language without nested patterns (the kind of programs produced by pattern-match compilation). This semantics for "unnested" Haskell could be extended simply to the full language by defining the meaning of a term with nested patterns to be the meaning of its compilation. For example, the meaning of `nodups1` would be identified with that of `nodups2`. Observe that this extended semantics relies on the ability to generate fresh names *within the semantics*. The implicit assumption in this approach that pattern-match compilation is just a semantically irrelevant elimination of syntactic sugar.

One defect of this extended semantics is that it is no longer compositional. A much more serious flaw, however, derives from the reliance on fresh name generation within the pattern-matching compilation. Recent developments[10, 4] in the semantics of staged languages reveal that the structural consequences of including name generation within a denotational semantics are considerable. This would have serious consequences for developing a simple logic for Haskell programs.