JFP **15** (6): 837–891, 2005. © 2005 Cambridge University Press
doi:10.1017/S0956796805005666 Printed in the United Kingdom

The logic of demand in Haskell

WILLIAM L. HARRISON
Department of Computer Science, University of Missouri, Columbia, MO, USA
(*e-mail:* `HarrisonWL@missouri.edu`)

RICHARD B. KIEBURTZ
Pacific Software Research Center, OGI School of Science & Engineering, Oregon Health & Science University

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

Although Haskell is known as a *lazy* functional language because of its default evaluation strategy, it contains a number of language constructs that force exceptions to that strategy. Among these features are pattern-matching, data type strictness annotations and the *seq* primitive. The semantics of pattern-matching are further enriched by *irrefutable pattern* annotations, which may be embedded within patterns. The interaction between Haskell's default lazy evaluation and its pattern-matching is surprisingly complicated. Although it offers the programmer a facility for *fine control of demand* (Harrison *et al.*, 2002), it is perhaps the aspect of the Haskell language least well understood by its community of users. In this paper, we characterize the control of demand first in a denotational semantics and then in a verification logic called "*P*-logic".

P-logic[1] is a modal logic based upon the familiar Gentzen-style sequent calculus (Girard, 1989). *P*-logic is expressive directly over Haskell expressions – the term language of the logic is Haskell 98. The two modalities of the logic, called weak and strong, determine whether a predicate is interpreted by a set of normalized values of its type (the strong interpretation) or by a set of computations of its type, which

[1] The name *P*-logic is taken from the Programatica project (`www.cse.ogi.edu/PacSoft/projects/programatica`) at OGI.

may or may not terminate (the weak interpretation). The strong modality is used to characterize properties of an expression occuring in a strict context of a program, or of an expression constructed in normal form. The weak modality can be used to characterize properties of an expression occuring in a non-strict context.

This paper introduces the fragment of P-logic that provides verification conditions for a core fragment of Haskell, including abstraction, application and case expressions (without guards). It also provides a self-contained description of a typed, denotational semantics for this Haskell fragment. The semantics for the Haskell fragment is based on an extension to the *type frames* semantics of the simply-typed lambda calculus (Gunter, 1992; Mitchell, 2000) and is closely related to an earlier treatment (Harrison *et al.*, 2002). This semantics constitutes the core of a denotational semantics for Haskell 98, the whole of which will be published in sequel articles.

Because Haskell patterns afford fine control of demand, it is not possible to give complete verification conditions for patterned abstractions or case expressions in a finite set of specific rules. In the presentation of P-logic, we give the logical inference rules for patterns by defining verification condition generators – functions on the term structure of patterns which construct *pattern predicates.* A verification condition for a property of a Haskell case branch is derived by applying a verification condition generator to its pattern and the list of predicates that its variables are assumed to satisfy. It generates a predicate characterizing terms that can match the pattern with its assumed properties. Verification condition generators are written as Haskell functions in a prototype implementation of P-logic.

The remainder of the paper proceeds as follows. Section 2 gives an overview of the Haskell fragment we consider here. This fragment contains the language constructs that most directly make use of pattern-matching. Section 3 contains background information for our semantic model: type frame semantics and the *simple model of ML polymorphism* (Ohori, 1989b; Ohori, 1989a). Section 4 summarizes the formal semantics of this fragment and section 5 presents the fragment of P-logic that deals with Haskell's fine control of demand. Example derivations in P-logic are also given in section 5. Soundness of the P-logic inference rules is established in section 6 and section 7 discusses some alternative approaches to verification logics. Section 8 summarizes our conclusions.

2 A Haskell fragment and its informal semantics

This section describes the fragment of Haskell we consider in this paper. This fragment, whose syntax is given in Figure 1, is representative of the Haskell constructs that depend on pattern-matching and strictness annotations. It constitutes a nearly-complete core language for Haskell expressions, omitting guarded expressions, type classes and overloaded operators. Section 2.2 gives an informal overview of the meaning of these constructs and section 2.2.6 discusses how fine control of demand, as specified in Haskell, entails complex evaluation strategies.

$$\begin{array}{lcl}
\textbf{type}\ \mathit{Name} & = & \mathit{String} \\
\textbf{data}\ \mathit{LS} & = & \mathit{Lazy} \mid \mathit{Strict}\ \textbf{deriving}\ \mathit{Eq} \\
\textbf{data}\ \mathit{Type} & = & \mathit{Name} \mid \mathit{Arrow}\ \mathit{Type}\ \mathit{Type} \mid \mathit{Name}\ [\mathit{Type}] \\
\textbf{data}\ P & = & \mathit{Pvar}\ \mathit{Name} \mid \mathit{Pcondata}\ \mathit{Name}\ [(\mathit{LS}, P)] \mid \mathit{Ptilde}\ P \mid \mathit{Ptuple}\ [P] \\
 & & \mid \mathit{Pwildcard} \\
\textbf{data}\ E & = & \mathit{Var}\ \mathit{Name} \mid \mathit{Constr}\ \mathit{Name}\ [(\mathit{LS},\ \mathit{Type})] \mid \mathit{Tuple}\ [E] \\
 & & \mid \mathit{Abs}\ \mathit{Name}\ E \mid \mathit{App}\ E\ E \mid \mathit{Case}\ E\ [(P, E)] \mid \mathit{Undefined}
\end{array}$$

Fig. 1. Abstract syntax of a Haskell fragment.

2.1 Data types

A data type declaration serves to define the data constructors of the type, giving the signature of each constructor as a series of type scheme arguments with optional strictness annotations. In the abstract syntax representation of a data constructor (see Fig. 1), strictness or non-strictness in each argument is explicitly designated by a tag of type *LS*. The signature of a data type, τ, is a finite set, Σ_τ whose elements are the abstract syntax terms designating the type signatures of the data constructors of the type. For example, from a data type declaration

$$\textbf{data}\ T\ \alpha_1 \cdots \alpha_n = \cdots \mid C\ \sigma_1 \cdots \sigma_k\ \mid \cdots$$

in which a k-place constructor, C, is declared without any strictness annotations, we obtain the signature element

$$\mathit{Constr}\ C\ \ [(\mathit{Lazy}, \sigma_1), \ldots, (\mathit{Lazy}, \sigma_k)] \in \Sigma_{T\ \alpha_1 \cdots \alpha_n}$$

Had any of the type arguments in the constructor declaration been given a strictness annotation, such as

$$T\ \alpha_1 \cdots \alpha_n = \cdots \mid C\ \ !\sigma_1 \cdots \sigma_k\ \mid \cdots$$

then in the signature element for the constructor, the tag *Strict* would accompany each of the listed type(s) that had been annotated in the declaration, i.e.

$$\mathit{Constr}\ C\ \ [(\mathit{Strict}, \sigma_1), \ldots, (\mathit{Lazy}, \sigma_k)] \in \Sigma_{T\ \alpha_1 \cdots \alpha_n}$$

The abstract syntax presented in Fig. 1 contains no direct representation of Haskell data type declarations. Instead, such data type declarations are part of the expression and pattern syntax, being encoded implicitly by the signatures of their individual constructors. This representation was chosen for its convenience in defining the semantics and logic of Haskell and could have been derived from a direct representation of data type declarations.

2.2 Case expressions

Patterns may occur in several different syntactic contexts in Haskell – in case branches, explicit abstractions, or on the left-hand sides of definitions[2]. Since the

[2] In a local (or `let`) definition, a pattern may occur as the entire left-hand side of an equation. A pattern used in this way is implicitly irrefutable, even if it is not prefixed by the character ($\sim$).

roles played by patterns are similar in each of these syntactic contexts, we shall focus on patterns in case branches.

2.2.1 *Evaluating case expressions*

A Haskell case expression is an instance of the syntactic schema:

$$\texttt{case}\ d\ \texttt{of}\ \{p_1 \rightarrow e_1;\ \cdots\ p_n \rightarrow e_n\}$$

in which d is an expression, which we call the *case discriminator*, and each of the $\{p_i \rightarrow e_i\}$ is a *case branch*, consisting of a pattern, p_i and an expression, e_i, called the *body* of the branch.

When a case expression is evaluated, the case discriminator is matched against the pattern of the first case branch. If the match succeeds, the body of the branch is evaluated in a context extended with the value bindings of pattern variables made by the match and returned as the value of the case expression. If the match fails, succeeding branches are tried until either one of the patterns matches or all branches have been exhausted. If no branch matches, then evaluation of the case expression fails with an unrecoverable error (i.e. it denotes bottom).

2.2.2 *Pattern matching is a binding operation*

A pattern fulfills two roles:

- **Control**: A case discriminator expression is evaluated to an extent sufficient 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.

2.2.3 *Variables and wildcard patterns*

A variable used as a pattern never fails to match; it binds to any the value of any term[3]. A value need not be normalized 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.

2.2.4 *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 data type declaration must be applied to exactly k sub-patterns when it is used in a pattern.

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

When a constructor occurs as the outermost 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 constructor is lazy in its i^{th} argument (i.e. its declaration has no strictness annotation at that argument), the argument is evaluated only if a value is required to match a corresponding sub-pattern. However, if the constructor is strict in its i^{th} argument position, a constructor application will evaluate the argument to head normal form, whether or not a value is required for pattern matching.

2.2.5 *Irrefutable patterns defer matching*

Matching a constructor pattern against a case discriminator expression evaluates the discriminator sufficiently to determine whether the pattern match succeeds. Haskell also contains the pattern annotation (~) for making pattern-matching lazier. If p is a pattern, then matching a case discriminator against $\sim p$ is deferred and the focus of computation proceeds to evaluate the body of the case branch. Annotating a pattern 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. We can say that irrefutable patterns are *control-disabled*.

2.2.6 *Fine control of demand: an example*

For example, with patterns constructed for the data type

```
data Tree = T Tree Tree | S Tree | L | R
```

we can construct the following case expressions:

```
case T L R of {T (S x) y -> y; T x y -> x}       evaluates to  L
case T L R of {T ~(S x) y -> y; T x y -> x}      evaluates to  R
case T L R of {T ~(S x) y -> x; T x y -> y}      evaluates to  error
case T L R of {~(T (S x) y) -> y; T x y -> x}    evaluates to  error
```

In the first of the 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 line, the embedded pattern `~(S x)` is control-disabled. The term `(T L R)` thus matches the pattern `(T ~(S x) y)`, binding `R` to the variable `y`. In the third line, 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 line illustrates that a deferred match of the term `(T L R)` against the pattern `(T (S x) y)` fails, although the match was evaluated to head normal form in response to a request for a binding for y alone.

3 Background

The denotational semantics for the Haskell fragment extends the type-frames semantics of the simply-typed lambda calculus (Gunter, 1992; Mitchell, 2000) to accomodate polymorphism and the structure required for modeling Haskell pattern-matching. Section 3.1 reviews type-frame semantics. Section 3.2 gives an overview of the model of polymorphism adopted here for the Haskell fragment: the *simple model of ML polymorphism* (Ohori, 1989b; Ohori, 1989a).

3.1 Type-frame semantics

One may think of a frame model as set-theoretic version of a cartesian closed category. That is, it provides "objects" (i.e. D_τ for each simple type τ) and axioms of representability and extensionality characterizing functions from objects to objects in terms of an application operator, $\bullet$. In this article, each simple type model D_τ is presumed to be built from sets with additional structure. We write $|D_\tau|$ for the underlying set of D_τ. We refer to D_τ as a ***frame object*** and to $|D_\tau|$ as its ***frame set***.

Definition 1
A ***frame*** is a pair $\langle \mathscr{D}, \bullet \rangle$ where

1. $\mathscr{D} = \{D_\tau \mid \tau \in \mathsf{Type} \ \& \ |D_\tau| \neq \emptyset\}$
2. $\bullet$ is a family of operations $\bullet_{\tau_1\tau_2} \in |D_{(\tau_1 \to \tau_2)}| \to |D_{\tau_1}| \to |D_{\tau_2}|$

Definition 2
The set function $\phi : |D_{\tau_1}| \to |D_{\tau_2}|$ is ***representable*** if

$$\exists f \in |D_{(\tau_1 \to \tau_2)}| \text{ s.t. } \phi(d) = f \bullet_{\tau_1\tau_2} d, \quad \forall d \in |D_{\tau_1}|$$

Definition 3
$\langle \mathscr{D}, \bullet \rangle$ is ***extensional*** if, for all $d \in |D_{\tau_1}|$, $f, g \in |D_{(\tau_1 \to \tau_2)}|$,

$$f \bullet_{\tau_1\tau_2} d = g \bullet_{\tau_1\tau_2} d \ \Rightarrow f = g$$

Definition 4
A value environment ρ is *compatible* with a (ground) type environment, $\mathscr{A}$, if

$$\forall x.\, x \in \mathsf{dom}(\rho) \Rightarrow \rho\, x \in |D_\tau|, \text{ where } (x : \tau) \in \mathscr{A}$$

The compatibility relation is designated by $\mathscr{A} \models \rho$. The set of value environments compatible with $\mathscr{A}$ is designated Env($\mathscr{A}$).

Definition 5
[Environment Model Condition]

Let $\langle \mathscr{D}, \bullet \rangle$ be any frame, $\lambda^{\rightarrow}$ be the simply-typed lambda calculus and ρ a value environment such that $\mathscr{A} \models \rho$. Then, the map $\mathscr{D}[\![-]\!] \in \lambda^{\rightarrow} \rightarrow Env \rightarrow (\bigcup |D_\tau|)$ obeys the ***environment model condition*** if the following equations hold:

$$\begin{array}{lcl} \mathscr{D}[\![\mathscr{A} \vdash x : \tau]\!]\rho & = & \rho\, x \\ \mathscr{D}[\![\mathscr{A} \vdash \lambda x.M : \tau_1 \rightarrow \tau_2]\!]\rho & = & f \quad \text{for unique } f \in |D_{\tau_1 \rightarrow \tau_2}| \\ \multicolumn{3}{c}{\text{such that } f \bullet d = \mathscr{D}[\![\mathscr{A}, x : \tau_1 \vdash M : \tau_2]\!]\rho[x \mapsto d] \text{ for all } d \in |D_{\tau_1}|} \\ \mathscr{D}[\![\mathscr{A} \vdash (M\ N) : \tau]\!]\rho & = & (\mathscr{D}[\![\mathscr{A} \vdash M : \tau' \rightarrow \tau]\!]\rho) \bullet (\mathscr{D}[\![\mathscr{A} \vdash N : \tau']\!]\rho) \end{array}$$

For any extensional frame $\mathscr{D}$, the above equations induce a model of the simply-typed lambda calculus (Gunter, 1992; Mitchell, 2000).

3.2 *A simple model of ML polymorphism*

The Girard–Reynolds calculus (alternately referred to as *System F* (Girard, 1972) and the *polymorphic lambda calculus* (Reynolds, 1974)) contains abstraction and application over types as well as over values. As such, it is sometimes referred to as a second-order lambda calculus. Denotational models of second-order lambda calculi exist (e.g., the PER model described in (Girard, 1989)). Such models provide one technique for specifying Haskell and ML polymorphism. Harper and Mitchell take this approach (Harper & Mitchell, 1993; Mitchell & Harper, 1988) for the core of Standard ML called core-ML. They translate a polymorphic core-ML term (i.e., one without type abstraction or application) into a second-order core-XML term (i.e., one with type abstraction or application). A core-ML term is then modeled by the denotation of its translation in an appropriate model of the second-order lambda calculus, core-XML.

ML polymorphism[4] is considerably more restrictive than the polymorphism expressible in a second-order lambda calculus; its types are of the form $\forall \alpha_0 \ldots \alpha_n.\sigma$ for a quantifier-free type scheme σ. Although outside the scope of this article, it appears that the Ohori model of polymorphism is adequate to the description of type classes in Haskell. However, we shall not consider type classes further in this article as they are not relevant to the issue we focus on here: the fine control of demand in Haskell.

Because of its restrictiveness relative to the Girard–Reynolds calculus, it is possible to give a predicative semantics to ML polymorphism (Ohori, 1989b; Ohori, 1989a) that is a conservative extension to the frame semantics of the simply-typed lambda calculus outlined in section 3.1. Ohori's model of ML polymorphism is particularly appealing because of its simplicity. It explains ML polymorphism in terms of simpler, less expressive things (such as the frame semantics of the simply-typed lambda calculus) rather than in terms of inherently richer and more expressive things (such as the semantics of the second-order lambda calculus).

[4] Following Ohori (Ohori, 1989b; Ohori, 1989a), we shall refer to the variety of polymorphism occuring in Haskell and ML as *ML polymorphism*. Both languages use varieties of Hindley–Milner polymorphism (Hindley, 1969; Milner, 1978).

We adopt Ohori's simple model of ML polymorphism (Ohori, 1989b; Ohori, 1989a) as part of the foundation for the Haskell fragment here. This model defines the meaning of polymorphic terms as type-indexed denotations of their ground instances (or *typings* as Ohori calls them). This approach to polymorphism factors the language specification into two parts: the specification of polymorphic terms (in Definition 12) and of their simply-typed instances (in Definitions 13 and 14).

Definition 6
A closed ML-polymorphic type ($\forall\alpha_1\ldots\alpha_n.\sigma$) is modeled by the type-indexed set of frame sets:

$$\{|D_\tau| \mid \tau = \sigma[\alpha_1/\tau_1,\ldots,\alpha_n/\tau_n], \tau_i \in \mathsf{Type}, \{\alpha_1,\ldots,\alpha_n\} = \mathrm{TV}(\sigma)\}$$

where Type is the set of all simple (i.e., ground) types and $\mathrm{TV}(\sigma)$ are the free type variables of σ.

Each core-ML polymorphic term is defined as the set of denotations of its ground instances, and these ground instances may be given a frame semantics in precisely the same manner as a simply-typed lambda calculus. Details of this model will be spelled out in greater detail in section 4.2.

4 Formal semantics of a Haskell fragment

This section presents the static and denotational semantics of the Haskell fragment. These are abstracted from the denotational semantics of Haskell (Harrison *et al.*, 2002) and are, for the most part, entirely conventional. The denotational semantics for the Haskell fragment is based on an extension to the type frames semantics of the simply-typed lambda calculus.

Because the focus of this article concerns the consequences of pattern-matching within the context of the Haskell language, much of this section is devoted to the necessary structure for modeling patterns. As the semantics developed here is a typed semantics (i.e., the terms defined denotationally are derivable typing judgments), we give a type system for patterns. This distinguishes our approach somewhat from other treatments of Haskell (Peyton Jones, 2003; Jones, 1999; Thompson, 1999; Hudak, 2000; Faxen, 2002) where patterns are not treated as first-class entities.

The static semantics for patterns associates a pattern with a type of the form $\sigma \rightarrow \varrho$, where σ is a conventional type scheme (i.e. constructed from type variables, type constants, $+$, $\times$, $\rightarrow$, and constructors arising from data type declarations) and ϱ is a record type. We introduce record types to capture statically the notion that a pattern produces a finite set of typed variable bindings when successfully matched against a value. Please note that incorporating record types in the semantic domain does not imply extending Haskell with record types, expressions, and values.

As noted in section 3.1, frames for the simply-typed lambda calculus consist of a pair, $\langle\mathscr{D},\bullet\rangle$, where $\mathscr{D}$ is a set containing the denotations of types and $\bullet$ is an application operator. The Haskell fragment presented here, being more expressive than the simply-typed lambda calculus, requires more structure to model with frames. We extend the notion of a type frame with structure including a partial order on the elements of frame sets, pointedness of frame objects, continuous functions that preserve order and limits, embedding-projection pairs for data types, the *Maybe* monad, and currying and uncurrying operations on functions. Formally, for the Haskell fragment, a frame is the tuple:

$$\langle \mathscr{D}, \bullet, \sqsubseteq, \sqcup, \bot, (-)^{\sharp}, (-)^{\flat}, c, c^{-1}, c^{M}, \mathit{Just}, \mathit{Nothing}, \texttt{>>=}, \mathit{return}, \mathit{lift}_{,}\oplus, \diamond, [\![\,]\!] \rangle$$

Here, $\sqsubseteq$, $\sqcup$, and $\bot$ are introduced to impose a pointed cpo structure on each of the frame objects $D_\tau \in \mathscr{D}$. Structure for embedding-projection pairs for data types, c and c^{-1}, represent the constructors introduced in data type declarations as well as the projections from values in data types. Curry $(-)^{\sharp}$ and uncurry $(-)^{\flat}$ operators are necessary to accomodate the view of "data constructors as functions" in Haskell. The *Maybe* monad and related structure are introduced to model pattern-matching; the structures

$$c^{M}, \mathit{Just}, \mathit{Nothing}, \texttt{>>=}, \mathit{return}, \mathit{lift}_{,}\oplus$$

are used for this purpose (and are described in detail below). Finally, control operators for both Kleisli composition ($\diamond$) and alternation ($[\![\,]\!]$) are introduced to model patterns and *case* expressions.

Such structures are the "bricks and mortar" of conventional denotational semantics and, in a domain-theoretic treatment, would be represented within some concrete domain structure. The frame semantics approach taken here axiomatizes this additional structure. These extended frames may be thought of as an abstraction of the cpo semantics of types which is the foundation of the semantics of functional programming languages (Schmidt, 1986; Gunter, 1992). The type frames for the Haskell fragment contain abstract operators corresponding to the concrete constructions (e.g., pointedness, embedding-projection pairs, etc.) that occur within domain theory, and semantically necessary properties of these abstract operators (e.g. extensionality, etc.) are characterized axiomatically. Suitable concrete, domain-theoretic representations of the extra structure in the frame semantics below have been suggested by several authors (Gunter, 1992; Schmidt, 1986; Mitchell, 2000; MacQueen *et al.*, 1984; Smyth & Plotkin, 1982).

Section 4.1 presents the type system for the Haskell fragment. Section 4.2 reviews the necessary definitions for relating polymorphic terms to their ground instances – these come directly from (Ohori, 1989b; Ohori, 1989a) and our treatment follows Ohori's closely. Section 4.2 defines the semantics of the polymorphic part of the Haskell fragment. The next two sections consider the frame semantics of the ground instances of the fragment. Section 4.3 presents the necessary extensions to the basic frame structure from Section 3.1 and Section 4.4 presents the semantic equations themselves.

4.1 The Haskell fragment

This section presents the type system for the Haskell fragment. The pattern class P does not include all varieties of Haskell patterns (e.g. "as" patterns or "$n+k$" patterns), while E includes *case* expressions without guards. These features have been omitted in the present treatment, however, as they are not relevant to Haskell's fine control of demand. In this section, we will write terms using Haskell's concrete syntax.

In Definitions 7 and 8 below, we formulate a type system for the Haskell fragment. The type system for this fragment is, except for the treatment of patterns, a conventional type system for implicit polymorphism. A typing judgment for an expression e is of the form $\Gamma \vdash e :: \sigma$, where any free variables occurring in the type scheme σ are implicitly quantified. To give a typed semantics for Haskell patterns, we must first give formal type rules for patterns. We shall use record types in the type rules for Haskell patterns. For these, we turn to Standard ML (Milner *et al.*, 1997) for inspiration. Patterns are given types of the form $(\sigma \rightarrow \varrho)$ where ϱ ranges over record types.

Definition 7

[Type language of the Haskell fragment] Below, b ranges over base types, α ranges over type variables, and T designates a type constructor assumed to be of arity n. There are simple types (ranged over by τ and referred to only as *types*) and type schemes (ranged over by σ). When a type scheme is used in a judgment, its free type variables are implicitly quantified.

Simple Types	$\tau \in \mathsf{Type}$	$::=$	$b \mid \tau \rightarrow \tau \mid T\ \underbrace{\tau \ldots \tau}_{n} \mid \zeta$
Type Schemes	$\sigma \in \mathsf{TypSch}$	$::=$	$\alpha \mid b \mid \sigma \rightarrow \sigma \mid T\ \underbrace{\sigma \ldots \sigma}_{n} \mid \varrho$
Simple record types	$\zeta \in sty$	$::=$	$\langle [styrow] \rangle$
Polymorphic record types	$\varrho \in pty$	$::=$	$\langle [ptyrow] \rangle$
Simple type rows	$styrow$	$::=$	$label :: \tau\ [, styrow]$
Polymorphic type rows	$ptyrow$	$::=$	$label :: \sigma\ [, ptyrow]$

Two record types are *disjoint* when they have no labels in common. We define an operation, $\otimes$, for combining disjoint record types such that

$$\begin{aligned} &\langle l_1 :: t_1, \ldots, l_n :: t_n \rangle \otimes \langle m_1 :: t_1, \ldots, m_k :: t_k \rangle \\ &\qquad = \langle l_1 :: t_1, \ldots, l_n :: t_n, m_1 :: t_1, \ldots, m_k :: t_k \rangle \end{aligned}$$

As we shall see in the type rules for patterns below in Definition 8, all record types arising in the typings of Haskell patterns are disjoint because Haskell patterns are *linear*; that is, Haskell does not allow repeated variables within patterns.

Definition 8

[Type Rules for Haskell Fragment] In the rules below, the notation Γ_x designates a type environment derived from Γ by removing any type binding for the variable x, should such exist in Γ.

Standard Rules

$$\frac{(x :: \sigma) \in \Gamma}{\Gamma \vdash x :: \sigma} \qquad \frac{\Gamma \vdash e_1 :: \sigma_2 \rightarrow \sigma_1 \quad \Gamma \vdash e_2 :: \sigma_2}{\Gamma \vdash e_1\, e_2 :: \sigma_1} \qquad \frac{\Gamma_x, x :: \sigma' \vdash e :: \sigma}{\Gamma \vdash \lambda x.e :: \sigma' \rightarrow \sigma}$$

$$\frac{\Gamma \vdash e :: \sigma' \quad \Gamma \vdash_{\text{pat}} p_i :: \sigma' \rightarrow \varrho_i \quad \Gamma + \varrho_i \vdash e_i :: \sigma}{\Gamma \vdash \mathsf{case}\ e\ \mathsf{of}\ \{p_1 \rightarrow e_1; \ldots ; p_n \rightarrow e_n\} :: \sigma}$$

where

$$\Gamma + \langle x_1 :: \sigma_1, \ldots, x_k :: \sigma_k \rangle = \Gamma_{x_1 \cdots x_k} \cup \{x_1 :: \sigma_1, \ldots, x_k :: \sigma_k\}$$

Patterns

$$\frac{}{\Gamma, x{::}\sigma \vdash_{\text{pat}} x :: \sigma \to \langle x{::}\sigma \rangle} \qquad \frac{\Gamma \vdash_{\text{pat}} p :: \sigma \to \varrho}{\Gamma \vdash_{\text{pat}} {\sim}p :: \sigma \to \varrho} \qquad \frac{}{\Gamma \vdash_{\text{pat}} \_ :: \alpha \to \langle\rangle}$$

$$\frac{(C{::}\sigma_1 \to \ldots \to \sigma_n \to \sigma) \in \Gamma \quad \Gamma \vdash_{\text{pat}} p_i :: \sigma_i \to \varrho_i \quad (1 \leqslant i \leqslant n)}{\Gamma \vdash_{\text{pat}} (C\ p_1 \ldots p_n) :: \sigma \to (\varrho_1 \otimes \ldots \otimes \varrho_n)}$$

$$\frac{\Gamma \vdash_{\text{pat}} p_i :: \sigma_i \to \varrho_i \quad (1 \leqslant i \leqslant n)}{\Gamma \vdash_{\text{pat}} (p_1, \ldots, p_n) :: (\sigma_1 \times \ldots \times \sigma_n) \to (\varrho_1 \otimes \ldots \otimes \varrho_n)}$$

The necessary technical vocabulary concerning type derivations for Ohori's model of polymorphism are presented below. An effort has been made to use Ohori's original terminology (Ohori, 1989b; Ohori, 1989a) whenever possible. None of these definitions are particularly surprising, although it is worth noting that, in our type language, free type variables within type schemes are implicitly quantified. Definition 9 presents the definition of what Ohori refers to as a *typing scheme*. This is, in more standard terminology, just a polymorphic term derivable in the type rules of Definition 8.

A *ground type assignment* $\mathscr{A}$ is a mapping from a finite set of term variables to Type. A *type scheme assignment* Γ is a mapping from a finite set of term variables to TypSch. A *substitution* is a function θ from type variables to TypSch s.t. $\theta\,\alpha \neq \alpha$ for only finitely-many type variables α. We designate the natural extension of a substitution θ to a map from TypSch to TypSch by θ^*. A *ground typing* is a judgment, $\mathscr{A} \vdash e :: \tau$, derivable in the rules of Definition 8. The Haskell terms defined by the semantics are precisely the *typing schemes* defined in Definition 9.

Definition 9
A formula, $\Gamma \vdash e :: \sigma$, is a ***typing scheme*** if, for all ground instances $(\mathscr{A}, \tau)$ of (Γ, σ), $\mathscr{A} \vdash e :: \tau$ is a ground typing. Furthermore, a typing scheme, $\Gamma \vdash e :: \sigma$, is ***polymorphic*** if σ contains a type variable.

4.2 Simple model of polymorphism for the Haskell fragment

Ohori's simple model of ML polymorphism defines the meaning of a polymorphic expression in terms of the type-indexed sets of denotations of its ground instances. It is a typed semantics, meaning that the denotations are given for derivable typing judgments of terms. We adopt this model of polymorphism for the Haskell fragment considered here.

Before delving into the technical details, we first present an intuitive example to motivate the approach. Consider the polymorphic term $(\emptyset \vdash \lambda x.x{::}\alpha \to \alpha)$. Any ground instance of this term (e.g. $\emptyset \vdash \lambda x.x{::}Int \to Int$) has a meaning within an appropriate frame $\mathscr{D}$. Within such a $\mathscr{D}$, if the elements of $|D_{\tau \to \tau'}|$ are actually functions from $|D_\tau|$ to $|D_{\tau'}|$, then the meaning of each of these instances is simply the identity function at its type, $id_{D_\tau} \in |D_{\tau \to \tau}|$. According to the simple model of polymorphism, the meaning of $(\emptyset \vdash \lambda x.x{::}\alpha \to \alpha)$ is just the set:

$$\{(\tau \to \tau, id_{D_\tau}) \mid \tau \in \mathsf{Type}\}$$

This example illustrates the structure of Ohori's model: a semantics of the ground typings of a language is extended conservatively to polymorphic terms. Extending the semantics of ground terms requires two additional data. Given a polymorphic term $(\Gamma \vdash e{::}\sigma)$, one must determine the appropriate ground type assignments $\mathscr{A}$ and ground types τ corresponding to Γ and σ, respectively. Definition 10 defines the set of ground type assignments compatible with a type assignment Γ that may contain free type variables, while Definition 11 specifies the set of ground types at which instances of a polymorphic term are defined. Definition 12 conservatively extends a semantics for the ground typings of the Haskell fragment to a semantics for polymorphic terms in the fragment.

Definition 10
The set of ***admissible type assignments under*** Γ is:

$$\mathsf{TA}(\Gamma) = \{\mathscr{A} \mid \exists\, \theta : FV(\Gamma) \rightarrow \mathsf{Type}.\ \mathscr{A} = \ \theta^* \circ \Gamma\ \}$$

Intuitively, $\mathscr{A} \in \mathsf{TA}(\Gamma)$ means that bindings in $\mathscr{A}$ are ground instances of the corresponding bindings in Γ. For example, suppose

$$\Gamma = (x \mapsto (\alpha \rightarrow Int)),\quad \mathscr{A}_0(x) = Int \rightarrow Int \text{ and } \mathscr{A}_1(x) = Char \times Int$$

Then $\mathscr{A}_0 \in \mathsf{TA}(\Gamma)$ but $\mathscr{A}_1 \notin \mathsf{TA}(\Gamma)$.

Definition 11
The set of types at which the polymorphic term $(\Gamma \vdash e :: \sigma)$ is defined is:

$$\begin{aligned}&\mathsf{Gr}(\Gamma \vdash e :: \sigma) \subseteq (\mathsf{dom}(\Gamma) \rightarrow \mathsf{Type}) \times \mathsf{Type}\\ &\mathsf{Gr}(\Gamma \vdash e :: \sigma) = \{(\mathscr{A}, \tau) \mid \exists\, \theta : FV(\Gamma) \rightarrow \mathsf{Type}.\ \mathscr{A} = \theta^* \circ \Gamma, \tau = \theta^* \sigma\}\end{aligned}$$

Several facts account for the well-definedness of Definition 12. Firstly, $\mathsf{Gr}(\Gamma \vdash e :: \sigma)$ may be considered as a mapping from ground type environments to Type because, for any derivable $\Gamma \vdash e :: \sigma$, there is a unique substitution $\theta : FV(\Gamma) \rightarrow \mathsf{Type}$ such that $\theta^*\Gamma = \mathscr{A}$ for any $\mathscr{A} \in \mathsf{dom}(\mathsf{Gr}(\Gamma \vdash e :: \sigma))$, where $\mathsf{dom}(-)$ is simply the first projection on a set of pairs. If $(\mathscr{A}, \tau), (\mathscr{A}, \tau') \in \mathsf{Gr}(\Gamma \vdash e :: \sigma)$, then $\tau = \theta^*\sigma = \tau'$. Secondly, we note that $\mathsf{dom}(\mathsf{Gr}(\Gamma \vdash e :: \sigma)) = \mathsf{TA}(\Gamma)$. And finally, we note that, if $\Gamma \vdash e :: \sigma$ is derivable, then so is any ground instance of it $\mathscr{A} \vdash e :: \tau$ (and $(\mathscr{A}, \tau) \in \mathsf{Gr}(\Gamma \vdash e :: \sigma)$).

Definition 12
[Semantics of The Haskell Fragment] Let $\mathscr{D}$ be any Haskell frame, $(\Gamma \vdash e :: \sigma)$ be a polymorphic term in the Haskell fragment, $\mathscr{A} \in \mathsf{TA}(\Gamma)$, and $\rho \in Env(\mathscr{A})$, then the following equation defines the semantics of polymorphic terms of the Haskell fragment as sets of type-indexed denotations of its ground instances:

$$\mathscr{D}[\![\Gamma \vdash e :: \sigma]\!]\, \mathscr{A}\, \rho = \{\, (\tau, \mathscr{D}[\![\mathscr{A} \vdash e :: \tau]\!]\, \rho) \mid \tau = (\mathsf{Gr}(\Gamma \vdash e :: \sigma))\mathscr{A}\,\}$$

The denotational definition of ground typings $\mathscr{D}[\![\mathscr{A} \vdash e :: \tau]\!]$ is presented in section 4.4.

4.3 *Haskell frames*

Recall that a frame for the Haskell fragment is a tuple:

$$\langle \mathscr{D}, \bullet, \sqsubseteq, \sqcup, \bot, (-)^{\sharp}, (-)^{\flat}, c, c^{-1}, c^{M}, \mathit{Just}, \mathit{Nothing}, \mathtt{>>=}, \mathit{return}, \mathit{lift}_{,} \oplus, \diamond, [\![\,]\!] \rangle$$

together with equations specifying properties of the elements. This section considers each of these additional structures in turn along with their properties.

4.3.1 CPO structure

The starting point for the frame semantics of Haskell is the cpo semantics of functional programming languages (Schmidt, 1986; Gunter, 1992; Mitchell, 2000). We assume that (ground) types are complete partial orders and that programs are continuous functions between them. A complete partial order (cpo) is a set S with a least element, $\bot$, and a partial order $\sqsubseteq$ such that every ascending chain, $\{x_i \in S \mid x_i \sqsubseteq x_{i+1}\}$, possesses a least upper bound in S, $\bigsqcup x_i$. A function f between cpos C and D is *continuous* if it is *monotonic* (i.e., $x \sqsubseteq_C y$ implies $f x \sqsubseteq_D f y$, for all $x, y \in C$) and it preserves least upper bounds of chains (i.e., $f(\bigsqcup c_i) = \bigsqcup(f c_i)$). The cpo semantics of the typed λ-calculus with recursion are frame models (Mitchell, 2000).

4.3.2 Pointedness in Haskell

Frames corresponding to Haskell types are necessarily *pointed* (i.e. they have a distinguished least element $\bot$) because of the need to solve recursive equations at all types. But Haskell's "lazy" pattern matching and the presence of the *seq* operator in the language puts further conditions on the bottom element of each frame. In the semantics of functional programming languages, the bottom elements in domains corresponding to constructed types ($T\ \tau_1 \ldots \tau_n$) (for type constructor T) are typically defined in terms of the domains denoting $\tau_1, \ldots, \tau_n$. That is, rather than introducing a new element, one constructs the bottom element $\bot_{(T\tau_1 \ldots \tau_n)}$ from the existing bottom elements $\bot_{\tau_1}, \ldots, \bot_{\tau_n}$. For example, logical choices for $\bot$ for the arrow, product, and list type constructors are:

$$\bot_{(\tau \to \tau')} \;=\; \lambda i.\, \bot_{\tau'} \qquad \bot_{(\tau \times \tau')} \;=\; (\bot_{\tau}, \bot_{\tau'}) \qquad \bot_{[\tau]} \;=\; (\bot_{\tau} {:} \bot_{[\tau]}) \tag{1}$$

The operator $\mathit{seq}{::}a \to b \to b$, one will recall, is strict in its first argument, so that $(\mathit{seq}\ e\ i)$ will be denoted by $\bot_{\tau'}$, if $i{::}\tau'$ and e is a Haskell term denoting $\bot_{\tau}$. Examples of terms that denote bottom are the Haskell terms *undefined*, (*error* "…"), and any non-terminating Haskell expression. Using *seq*, Haskell programs may distinguish bottom-denoting terms from the definitions given in (1) above. Examples illustrating this distinction are presented in Table 1. For this example, we have chosen to typify any $\bot$-denoting Haskell expression by *undefined*. Evaluating *pair1*, *fun1*, and *intlist1* will all produce errors, because *seq*, being strict in its first argument, evaluates the expression *undefined*. Evaluating *pair2*, *fun2*, and *intlist2* all produce the integer 1, because terms corresponding to (1) – *pairBot*, *funBot*, and *intlistBot* – do not denote $\bot$ in their respective types. Since $\mathit{pair1} \neq \mathit{pair2}$, $\mathit{fun1} \neq \mathit{fun2}$, and $\mathit{intlist1} \neq \mathit{intlist2}$,

Table 1. *The Haskell programs in the right column correspond to the standard domain-theoretic constructions of* $\perp$*, while those in the left column use the* $\perp$*-denoting Haskell term "undefined." The Haskell seq operator distinguishes the two*

— $\perp$-denoting Haskell terms			— standard constructions of $\perp$		
hPairBot	::	(Int, Int)	*pairBot*	::	(Int, Int)
hPairBot	=	*undefined*	*pairBot*	=	(*undefined*, *undefined*)
hFunBot	::	$Int \rightarrow Int$	*funBot*	::	$Int \rightarrow Int$
hFunBot	=	*undefined*	*funBot*	=	$\lambda x.$*undefined*
hIntListBot	::	$[Int]$	*intlistBot*	::	$[Int]$
hIntListBot	=	*undefined*	*intlistBot*	=	*undefined* : *undefined*
discern x	=	*seq x* 1	*fun1*	=	*discern hFunBot*
			fun2	=	*discern funBot*
pair1	=	*discern hPairBot*	*intlist1*	=	*discern hIntListBot*
pair2	=	*discern pairBot*	*intlist2*	=	*discern intlistBot*

the standard domain constructions of $\perp$ given in (1) are untenable for the Haskell language.

A consequence of the inclusion of *seq* in Haskell is that we must provide axioms specifying the difference between the constructions of (1) and the bottom element in $|D_\tau|$. In particular, each of the standard constructions in (1) must be strictly above the bottom element in its frame:

$$\begin{array}{lcl} \perp_{(\tau\times\tau')} & \sqsubset & (\perp_\tau, \perp_{\tau'}) \\ \perp_{(\tau\rightarrow\tau')} & \sqsubset & \lambda x.\ \perp_{\tau'} \\ \perp_{[\tau]} & \sqsubset & (\perp_\tau : \perp_{[\tau]}) \end{array}$$

Furthermore, there must be no elements "in between":

$$\begin{array}{lcl} x \sqsubseteq (\perp_\tau, \perp_{\tau'}) & \Rightarrow & (x = \perp_{(\tau\times\tau')}) \vee (x = (\perp_\tau, \perp_{\tau'})),\ \text{for all } x \in |D_{(\tau\times\tau')}| \\ x \sqsubseteq \lambda x.\ \perp_{\tau'} & \Rightarrow & (x = \perp_{(\tau\rightarrow\tau')}) \vee (x = \lambda x.\ \perp_{\tau'}),\ \text{for all } x \in |D_{(\tau\rightarrow\tau')}| \\ x \sqsubseteq (\perp_\tau : \perp_{[\tau]}) & \Rightarrow & (x = \perp_{[\tau]}) \vee (x = (\perp_\tau : \perp_{[\tau]})),\ \text{for all } x \in |D_{[\tau]}| \end{array}$$

4.3.3 Currying and uncurrying

We assume the existence of operators *curry* and *uncurry*:

$$(\_)^\sharp \in |D_{(\tau_1\times\ldots\times\tau_n\rightarrow\tau)\rightarrow(\tau_1\rightarrow\ldots\rightarrow\tau_n\rightarrow\tau)}|$$
$$(\_)^\flat \in |D_{(\tau_1\rightarrow\ldots\rightarrow\tau_n\rightarrow\tau)\rightarrow(\tau_1\times\ldots\times\tau_n\rightarrow\tau)}|$$

The curry and uncurry operators in each frame obey the following equations:

$$\begin{array}{rcl} (f^\flat)^\sharp & = & f,\ \text{for } f \in |D_{\tau_1\rightarrow\ldots\rightarrow\tau_n\rightarrow\tau}| \\ (g^\sharp)^\flat & = & g,\ \text{for } g \in |D_{\tau_1\times\ldots\times\tau_n\rightarrow\tau}| \end{array}$$

Only the curry operator is used explicitly in this article (to define Haskell constructors as curried functions), but the uncurry operator $(-)^{\flat}$ is needed for the axiomatization of $(-)^{\sharp}$.

4.3.4 Haskell data types

Data types in Haskell may be recursive and this, combined with Haskell's laziness, allows for the construction of infinite data values. Pattern-matching in Haskell, however, is based upon only a finite portion of a structured value in a data type. While the semantic framework presented in this section allows for solution of the recursive domain equations associated with data type declarations the issue of infinite data values is not germane to pattern-matching. To meet the goals of the present article, we do not need to illustrate a model of recursive data types and have therefore chosen to omit it.

In a Haskell data type declaration, a programmer can write strictness annotations on the type arguments to constructors. For example, the data type declared by:

$$\textbf{data}\ Foo\ =\ S\ !Int\ Bool$$

has a single binary constructor that has a strictness annotation on its first argument. The function denoted by the constructor S is semantically equivalent to the abstraction:

$$\lambda x.\lambda y.seq\ x\ (S\ x\ y)$$

Note that many Haskell programmers might call this function "strict in its first argument" meaning that the *saturated* application $(S\, e_1\, e_2)$ will denote $\bot$ if e_1 denotes $\bot$. However, the use of the word "strict" to describe the constructor S conflicts with the usual meaning of the term in denotational semantics (Gunter, 1992; Mitchell, 2000). Considered as a function, S being strict in its first argument would mean that the following equation holds: $S\ \bot_{Int} = \bot_{(Bool \to Foo)}$. Note however that $(S\ \bot_{Int})$ is semantically equivalent to the abstraction, $(\lambda y.seq\ undefined\ (S\ undefined\ y))$, which is not denoted by $\bot_{(Bool \to Foo)}$ as noted in section 4.3.2. The two notions of strictness could not be distinguished if Haskell lacked the *seq* operator. We will refer to a function $f \in |D_{\tau_1 \to \ldots \to \tau_n \to b}|$ (where type b is a base type) as *saturation strict* (or "sat-strict" for short) in its i-th argument, if $(f \bullet v_1 \ldots \bullet v_n) = \bot_b$ whenever $v_i = \bot_{\tau_i}$.

4.3.5 Type frames for data constructors

Consider the Haskell data type declaration:

$$\textbf{data}\ T\ \alpha_1 \ldots \alpha_n\ =\ \ldots \mid (C_i\ \sigma_{i,1} \ldots \sigma_{i,k_i}) \mid \ldots \tag{2}$$

where $\bigcup FV(\sigma_{i,j}) \subseteq \{\alpha_1, \ldots, \alpha_n\}$. Following Definition 6, the denotation of this type is a set of frame objects of the form $D_{(T\ \tau_1 \ldots \tau_n)}$. What is the structure of these $D_{(T\ \tau_1 \ldots \tau_n)}$? For each constructor C_i, we introduce the following family of functions into the frame model:

$$\begin{array}{lcl} c_i & \in & |D_{(\tau_{i,1} \times \ldots \times \tau_{i,k_i}) \to (T\ \tau_1 \ldots \tau_n)}| \\ c_i^{-1} & \in & |D_{(T\ \tau_1 \ldots \tau_n) \to (\tau_{i,1} \times \ldots \times \tau_{i,k_i})}| \end{array}$$

Note that a Haskell constructor is a curried function corresponding to $c_i^\sharp$. Equations (3) and (4) specify that c_i and c_i^{-1} form an *embedding-projection pair* (Gunter, 1992; Mitchell, 2000):

$$
\begin{aligned}
c_i^{-1} \circ c_i &= id_{D_{\tau_1 \times \ldots \times \tau_n}} && (3)\\
c_i \circ c_i^{-1} &\sqsubseteq id_{D_T} && (4)\\
c_i^{-1} \bullet (c_j \bullet \vec{v}) &= \bot_{\tau_1 \times \ldots \times \tau_n} \ \text{ for } i \neq j && (5)
\end{aligned}
$$

In Equation (5), C_j is a T constructor distinct from C_i and c_j is its corresponding frame function. Let S_i be the set of indices of arguments for constructor C_i that are declared with the strictness annotation "!". Then,

$$
\begin{aligned}
\bot_T &\sqsubset c_i \bullet (v_{i,1}, \ldots, v_{i,k_i}) \text{ where } v_{i,l} \in |D_{\tau_{i,l}}| \text{ and } (v_l = \bot_{\tau_{i,l}} \Leftrightarrow l \notin S_i) && (6)\\
\bot_T &= c_i \bullet (v_{i,1}, \ldots, v_{i,k_i}) \text{ where, for at least one } l \in S_i,\ v_l = \bot_{\tau_{i,l}} && (7)
\end{aligned}
$$

These equations determine when the bottom element in the data type T is separated from the bottom elements of arguments of a constructor application and when the bottom elements are coalesced. Note that, if C_i is declared without strictness annotations, which is the default in a Haskell program, then (6) and (7) simplify to:

$$\bot_T \sqsubset c_i \bullet (\bot_{\tau_{i,1}}, \ldots, \bot_{\tau_{i,k_i}})$$

4.3.6 Representing the Maybe monad in $\mathscr{D}$

The semantics of Haskell pattern-matching will be presented as a computation in the Maybe monad (Harrison *et al.*, 2002). We must consider the representation of such a monadic computation in the frame semantics. A computation in the Maybe monad is coded in the data type whose Haskell declaration is:

$$\textbf{data}\ Maybe\ \alpha \ = \ Just\ \alpha \mid Nothing$$

Following section 3.2, this polymorphic type is denoted by the following set:

$$\{D_{(Maybe\,\tau)} \mid \tau \in \mathsf{Type}\}$$

Furthermore, there are the families of functions corresponding to the *Maybe* constructors:

$$
\begin{aligned}
&Just_\tau \in |D_{\tau \to Maybe\,\tau}| \qquad && Nothing_\tau \in |D_{Maybe\,\tau}|\\
&Just_\tau^{-1} \in |D_{Maybe\,\tau \to \tau}| &&
\end{aligned}
$$

Because it is an essential part of the semantics of pattern-matching, we coin the name $purify_\tau$ for $Just_\tau^{-1}$. It is so named because it projects a computation into a pure value. The actions of *purify* are given by equations (3) and (5) above:

$$
\begin{aligned}
purify_\tau \bullet (Just_\tau \bullet v) &= v \ \text{ for any } v \in |D_\tau|\\
purify_\tau \bullet Nothing_\tau &= \bot_\tau
\end{aligned}
$$

Functions for the unit and bind of the *Maybe* monad – written as `return` and (>>=) in concrete Haskell syntax – are also added. The syntax for the *Maybe* monad within the Haskell frame mirrors the concrete syntax of Haskell, but the reader should distinguish the two.

$$\begin{array}{lcl} return_\tau & \in & |D_{\tau \to Maybe\,\tau}| \\ return_\tau & = & Just_\tau \end{array} \qquad \begin{array}{lcl} \texttt{>>=}_{\tau,\tau'} \in |D_{Maybe\,\tau \to (\tau \to Maybe\,\tau') \to Maybe\,\tau'}| & & \\ (Just_\tau\, v)\ \texttt{>>=}_{\tau,\tau'}\ f & = & f \bullet v \\ Nothing_\tau\ \texttt{>>=}_{\tau,\tau'}\ f & = & Nothing_{\tau'} \end{array}$$

The following three equations express the laws required of a monad. The particular formulation we use is sometimes refered to as the *Kleisli* formulation of monads (Barr & Wells, 1990). The third equation specifies the transitivity of the bind operation:

$$\begin{array}{lcl} (\texttt{>>=}_{\tau,\tau'}\ f) \circ return_\tau & = & f \\ (\texttt{>>=}_{\tau,\tau}\ return_\tau) & = & id_{Maybe\,\tau} \\ (\texttt{>>=}_{\tau_1,\tau_2} g) \circ (\texttt{>>=}_{\tau_0,\tau_1} f) & = & \texttt{>>=}_{\tau_0,\tau_1}((\texttt{>>=}_{\tau_1,\tau_2} g) \circ f) \end{array}$$

Here, $(\texttt{>>=}_{\tau,\tau'}\ f)$ is a right section of the binary infix operator, >>=.

Furthermore, for a data type T, defined as above in (2), the projections associated with its constructors may be factored through a computational version of the projection c_i^M:

$$\begin{array}{lcl} c_i^M \in |D_{(T\,\tau_1 \ldots \tau_n) \to Maybe(\tau_{i,1} \times \ldots \times \tau_{i,k_n})}| & & \\ c_i^M \bullet \perp_{(T\,\tau_1 \ldots \tau_n)} & = & \perp_{Maybe(\tau_{i,1} \times \ldots \times \tau_{i,k_n})} \\ c_i^M \bullet (c_i \bullet (v_{i,1}, \ldots, v_{i,k_i})) & = & Just \bullet (v_{i,1}, \ldots, v_{i,k_i}) \ \text{ for } c_i \bullet (v_{i,1}, \ldots, v_{i,k_i}) \neq \perp \\ c_i^M \bullet (c_j \bullet (v_{j,1}, \ldots, v_{1,k_j})) & = & Nothing \qquad (i \neq j) \\ c_i^{-1} & = & purify \circ c_i^M \end{array}$$

4.3.7 *Frame semantics of records*

Records have a constructed bottom just as other data types do (see section 4.3.2) according to the pointwise ordering. We refer to the constructed bottom of ζ, $\langle m_0 = \perp_{t_0}, \ldots, m_n = \perp_{t_n} \rangle$, as $\langle \perp \rangle_\zeta$. We then define a function, $lift_\zeta$, which plays a crucial rôle in defining the meaning of the irrefutable pattern:

$$\begin{array}{lcl} lift_\zeta : |D_{(Maybe\,\zeta)}| \to |D_{(Maybe\,\zeta)}| & & \\ lift_\zeta\ Nothing_\zeta & = & Just_\zeta\ \langle \perp \rangle_\zeta \\ lift_\zeta\ (Just_\zeta\ r) & = & Just_\zeta\ r \\ lift_\zeta\ \perp_{(Maybe\,\zeta)} & = & \perp_{(Maybe\,\zeta)} \end{array}$$

We require an operator to combine a tuple of records computed in the Maybe monad into a computation of a single record. For every tuple of record types, $\zeta_1, \ldots, \zeta_n$, with non-overlapping field names, we define the following operation. Subscripts may be omitted when clear from context.

$$\oplus_{\zeta_1 \times \ldots \times \zeta_n} : |D_{(Maybe \zeta_1 \times \ldots \times Maybe \zeta_n)}| \to |D_{Maybe(\zeta_1 \otimes \ldots \otimes \zeta_n)}|$$

$$\oplus (m_1, \ldots, m_n) \ = \ \begin{cases} Nothing_{(\zeta_1 \otimes \ldots \otimes \zeta_n)} & \text{if } \exists i \in [1..n].\ m_i = Nothing_{\zeta_i} \\ Just_{(\zeta_1 \otimes \ldots \otimes \zeta_n)} r & \text{if } \forall i \in [1..n].\ m_i = Just_{\zeta_i} r_i \end{cases}$$

where r is the record whose fields are: $f = v \in r \Leftrightarrow \exists i \in [1..n].\ f = v \in r_i$. When applied to a tuple of record-typed computations in the *Maybe* monad, the operator $\oplus$ returns *Nothing* when any of the tuple components is *Nothing*.

4.3.8 *Control operators: Diagrammatic Kleisli* ($\diamond$) *and Alternation* ($[\!])$)

For any simple types τ, τ_1, τ_2, and τ_3, we have the operators ($\diamond$) and ($[\!]$):

$$(\diamond) : |D_{(\tau_1 \rightarrow Maybe\,\tau_2)}| \rightarrow |D_{(\tau_2 \rightarrow Maybe\,\tau_3)}| \rightarrow |D_{(\tau_1 \rightarrow Maybe\,\tau_3)}|$$
$$([\!]) \in |D_{(Maybe\,\tau) \rightarrow (Maybe\,\tau) \rightarrow (Maybe\,\tau)}|$$

These operators are defined by the following equations:

$$f \diamond g \;=\; \lambda x.(f\,x) >\!\!>\!= g \qquad \begin{array}{rcl} Nothing_\tau \; [\!] \; y & = & y \\ (Just_\tau\, v)\; [\!] \; y & = & Just_\tau\, v \\ \bot_{(Maybe\,\tau)} \; [\!] \; y & = & \bot_{(Maybe\,\tau)} \end{array}$$

4.4 *Typed semantics for the simply-typed Haskell fragment*

As noted in section 4.2, the denotations of polymorphic terms we have chosen for the Haskell fragment is a conservative extension of the semantics of ground terms. This section presents a frame semantics for ground typings of the Haskell fragment. Definitions 12, 13, and 14 constitute the semantics of the Haskell fragment. Please note that we drop the "$\mathscr{D}$" from the semantic function for $[\![-]\!]$ in the remainder of the article.

Definition 13 (*Typed Semantics for Patterns*)
Let $\mathscr{A} \vdash_{\text{pat}} p :: \tau \rightarrow \zeta$ be a derivable typing of pattern p, then the *typed semantics for the pattern fragment* is:

$$[\![\mathscr{A} \vdash_{\text{pat}} p :: \tau \rightarrow \zeta]\!] \in |D_{\tau \rightarrow Maybe\,\zeta}|$$

Equations[5] defining $[\![\mathscr{A} \vdash_{\text{pat}} p :: \tau \rightarrow \zeta]\!]$ are:

$$\begin{array}{rcl}
[\![\mathscr{A} \vdash_{\text{pat}} x :: \tau \rightarrow \zeta]\!] & = & return\,\langle x = -\rangle \\
& \text{where} & \langle x = -\rangle \in |D_{\tau \rightarrow \langle x::\tau\rangle}| \\
& & \langle x = -\rangle \bullet v = \langle x = v\rangle \\
[\![\mathscr{A} \vdash_{\text{pat}} \_ :: \tau \rightarrow \zeta]\!] & = & return\,\langle -\rangle \\
& \text{where} & \langle -\rangle \in |D_{\tau \rightarrow \langle\rangle}| \\
& & \langle -\rangle \bullet v = \langle\rangle \\
[\![\mathscr{A} \vdash_{pat} (C\ p_1 \ldots p_n) :: \tau \rightarrow \zeta]\!] & = & c^M \diamond (return \circ (m_1 \times \ldots \times m_n)) \diamond \oplus \\
[\![\mathscr{A} \vdash_{pat} (p_1, \ldots, p_n) :: \tau \rightarrow \zeta]\!] & = & (return \circ (m_1 \times \ldots \times m_n)) \diamond \oplus \\
& \text{where} & f_1 \times \ldots \times f_n \;=\; \lambda(x_1, \ldots, x_n).(f_1\,x_1, \ldots, f_n\,x_n) \\
& & m_i \;=\; [\![\mathscr{A} \vdash_{pat} p_i :: \tau_i \rightarrow \zeta_i]\!] \\
[\![\mathscr{A} \vdash_{\text{pat}} \sim p :: \tau \rightarrow \zeta]\!] & = & lift_\zeta \circ [\![\mathscr{A} \vdash_{\text{pat}} p :: \tau \rightarrow \zeta]\!]
\end{array}$$

[5] We remind the reader that Kleisli composition ($\diamond$) is written diagrammatic order, while function composition ($\circ$) is in applicative order.

Definition 14 (*Typed Semantics of Ground Typings*)
Let $\mathscr{A} \vdash e :: \tau$ be a derivable ground typing and ρ be a value environment compatible with $\mathscr{A}$; then the *typed semantics for the expression fragment* is:

$$[\![\mathscr{A} \vdash e :: \tau]\!]\rho \in |D_\tau|$$

The equations defining $[\![\mathscr{A} \vdash e :: \tau]\!]$ are:

$$[\![\mathscr{A} \vdash \lambda x.e :: \tau \to \tau']\!]\rho \;=\; f$$
$$\text{where } f \bullet v = [\![\mathscr{A}, x{::}\tau \vdash e :: \tau']\!]\rho[x \mapsto v],\; \text{for all } v \in |D_\tau|$$

$$[\![\mathscr{A} \vdash e_1\, e_2 :: \tau]\!]\rho \;=\; ([\![\mathscr{A} \vdash e_1 :: \tau' \to \tau]\!]\rho) \bullet ([\![\mathscr{A} \vdash e_2 :: \tau']\!]\rho)$$

$$[\![\mathscr{A} \vdash \mathsf{case}\ e\ \mathsf{of}\ \{p_1 \to e_1; \ldots; p_n \to e_n\} :: \tau]\!]\rho \;=\; \mathit{purify} \bullet ((m_1 \bullet \varepsilon)\, [\!]\, \ldots\, [\!]\, (m_n \bullet \varepsilon))$$

where

$$\begin{aligned}
&\varepsilon = [\![\mathscr{A} \vdash e :: \tau']\!]\rho \\
&m_i = [\![\mathscr{A} \vdash_{\text{pat}} p_i :: \tau_i \to \zeta_i]\!] \;\diamond\; (\lambda r.\ \mathit{return}([\![\mathscr{A} + \zeta_i \vdash e_i :: \tau_i]\!](\rho + r))) \\
&\rho + \langle x_1 {=} v_1, \ldots, x_k {=} v_k \rangle = \rho[x_1 \mapsto v_1, \ldots, x_k \mapsto v_k] \\
&\mathscr{A} + \langle x_1{::}\tau_1, \ldots, x_k{::}\tau_k \rangle = \mathscr{A} \cup \{x_1{::}\tau_1, \ldots, x_k{::}\tau_k\}
\end{aligned}$$

$$[\![\mathscr{A} \vdash \mathit{undefined} :: \tau]\!]\rho \;=\; \bot_\tau$$

5 Logic for the Haskell fragment

While the denotational semantics defines a meaning for expressions in terms of an abstract model, a verification logic expresses static assertions about semantic properties of expressions. An assertion in P-logic takes the form of an n-ary predicate applied to n terms. There is a distinguished predicate symbol (===) that denotes semantic equality of terms. Reasoning in the logic is based upon a set of proof rules, each relating a consequent assertion to a set of possibly simpler antecedents, called the *verification conditions* for the consequent. If a rule is sound, the truth of its verification conditions is a logically sufficient condition to assure the truth of its consequent.

P-logic is useful both for equational reasoning about expressions in a Haskell program and for reasoning about properties other than equality. Examples of such properties are that an expression denotes a non-bottom value in its type, or that a *list*-typed expression denotes a finite list, or that an *Integer*-typed expression denotes a non-zero value.

In this section, our principal goal will be to give meaning to formulas of P-logic by relating them to the formal semantics of the Haskell fragment. In particular, we shall prove the soundness of some basic proof rules of P-logic by showing that the logical implications stated by these rules are valid when formulas of the logic are interpreted in a frame semantics for the Haskell fragment.

Formalizing the semantics of all of P-logic and proving soundness of its inference rules is a formidable task, far too much to describe in a single journal article, and one we have not yet completed. P-logic has many predicate forms, including recursively-defined predicates, predicates that express properties of monadic computations and predicates derived from sections of boolean operators, that are not mentioned here. Here, we have focused on formalizing an essential core of P-logic, limiting the scope to predicates that assert elementary properties of expressions in the core Haskell fragment.

We describe here only unary predicates in P-logic. The treatment of multi-place predicates (including equality) presents no fundamental difficulty but the formal notation needed to express the semantics of multi-place predicates is necessarily heavier.

A unary predicate $P :: \mathit{Pred}\ \tau$ characterizes a set of terms of type τ. A slogan to keep in mind is that *unary predicates refine types*. The typing of a predicate formula in a simple typing environment, $\mathscr{A}$, is derived from the typings of term constants and data constructors that occur in the formula. Predicates, like terms, may be polymorphically typed. In particular, the predicate constant, Univ, has the universal type $\mathit{Pred}\ \alpha$, where α is a free type variable.

Informally, a well-typed term satisfies a compatibly-typed predicate if the denotation of the term belongs to the set denoted by the predicate. We shall formalize this notion in section 5.7. We write $e :: \tau ::: P$ for the assertion that a term e satisfies predicate P at type τ. Often, explicit typing will be omitted when stating rules of P-logic, as suitable, generic types can be inferred from contexts.

Because function and data constructor applications are non-strict by default in Haskell's evaluation semantics, two notions of the strength of a predicate are sensible. The interpretation of a predicate may be explicitly restricted by prefixing it with the modal operator (\$), to designate the *strong* modality of P-logic. A strong predicate, $\$P :: \mathit{Pred}\ \tau$, is satisfied by a term, $e :: \tau$, in value environment ρ if both e satisfies P and in addition, the denotation of e with respect to ρ is not the bottom element in D_τ. By convention, a predicate is interpreted in the weak modality if it is not explicitly strengthened.

In this section we give a brief introduction to the fragment of P-logic that is relevant to pattern-matching in Haskell. The rules have been expressed in terms of Haskell's surface syntax insofar as possible. However, to express logical rules involving patterns we shall employ some algorithms that are more clearly written using abstract syntax for Haskell expressions. In particular, strictness annotations that may accompany the declaration of a data constructor are not apparent in the concrete syntax of a constructor application. The abstract syntax for a data constructor (see Fig. 1) manifests its strictness properties.

5.1 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 denotation is bottom.

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

$$\overline{\vdash \mathsf{Univ} :: \mathit{Pred}\ \sigma} \qquad \overline{\vdash \mathsf{UnDef} :: \mathit{Pred}\ \sigma}$$

$$\frac{\vdash P_1 :: \mathit{Pred}\ \sigma_1 \quad \cdots \quad \vdash P_k :: \mathit{Pred}\ \sigma_k}{C^{(k)} :: \sigma_1 \to \cdots \to \sigma_k \to \sigma \vdash C^{(k)}\ P_1 \cdots P_k :: \mathit{Pred}\ \sigma}$$

$$\frac{\vdash P_1 :: \mathit{Pred}\ \sigma_1 \quad \cdots \quad \vdash P_k :: \mathit{Pred}\ \sigma_k}{\vdash (P_1, \ldots, P_k) :: \mathit{Pred}\ (\sigma_1, \ldots, \sigma_k)} \qquad \frac{\vdash P :: \mathit{Pred}\ \sigma_1 \quad \vdash Q :: \mathit{Pred}\ \sigma_2}{\vdash P \to Q :: \mathit{Pred}\ (\sigma_1 \to \sigma_2)}$$

$$\frac{\vdash P :: \mathit{Pred}\ \sigma}{\vdash \neg P :: \mathit{Pred}\ \sigma} \qquad \frac{\vdash P :: \mathit{Pred}\ \sigma}{\vdash \$P :: \mathit{Pred}\ \sigma}$$

Fig. 2. Predicate typing rules.

```
data Pr  =  Univ                  {- the Universal predicate -}
         |  UnDef                 {- the Undefined predicate -}
         |  ConPred Name [Pr]     {- pattern predicate -}
         |  Strong Pr             {- strengthened predicate -}
         |  PredVar Name          {- predicate variable -}
         |  PArrow Pr Pr          {- arrow predicates -}
         |  TuplePred [Pr]        {- tuple predicate -}
         |  Pneg Pr               {- negated predicate -}
```

Fig. 3. Abstract syntax of predicates as a Haskell data type.

1. As predicates refine types, a type constructor may be implicitly lifted to act as a predicate constructor. For example, given a predicate $P :: \mathit{Pred}\ \tau$, we can form a new predicate $\mathit{Maybe}\ P :: \mathit{Pred}\,(\mathit{Maybe}\ \tau)$.
2. An N-tuple of predicates forms a predicate which is satisfied by a tuple of N Haskell expressions that satisfy the tupled predicates, pointwise.
3. The data constructors declared for data types 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 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. A nullary data constructor, such as `[]`, is lifted to a predicate constant.
4. The "arrow" predicate constructor is used to compose predicates that express properties of functions. An arrow predicate $P \to Q$ is satisfied by a function-typed expression, e, if given any argument expression e' that satisfies P, the application $(e\,e')$ satisfies Q. We refer to P as the domain predicate and Q as the codomain predicate of the arrow predicate, $P \to Q$.

Figure 2 gives typing rules for predicates. Figure 3 contains a Haskell definition of the abstract syntax for the P-logic predicate language.

5.2 Judgment forms

A judgment form in P-logic is a relation of three components:

- a typing environment, Γ;
- a list of zero or more assertions, Π, whose conjunction is an assumption supporting the judgment;
- a list of zero or more assertions, Δ, whose disjunction constitutes the conclusion of the judgment.

A judgment form is written in sequent notation as $\Gamma; \Pi \vdash_{\mathscr{P}} \Delta$. When the typing environment is superfluous, as when unambiguous types of expressions and predicates can be inferred from their structure, we shall omit the typing environment from the sequent notation, writing just $\Pi \vdash_{\mathscr{P}} \Delta$.

For example, we can express a property of the function *map*, defined in Haskell's standard prelude, with the sequent

$$f ::: P \to Q \vdash_{\mathscr{P}} map\ f ::: \$(\$[P] \to \$[Q])$$

In this sequent, the function symbol, f, denotes a partial function from arguments with property P to results with property Q. (Recall from Figure 1 that the bottom element of an arrow type in Haskell's semantics is distinct from the standard construction of bottom as a function.) The conclusion of the sequent asserts that *map f* also denotes a function which, when applied to a normal[6] list whose elements have the property P, yields a normal list whose elements have the property Q. A judgment formed with unary predicates resembles a typing judgment; as noted, unary predicates refine types.

5.3 Inference rules for properties of the Haskell fragment

Inference rules of P-logic are written as relations among judgment forms. A rule is a relation between zero or more antecedents and a single consequent judgment. In sequent calculus style, each term-specific rule introduces a property associated with a particular term construction into the consequent judgment. A rule may introduce such a property either on the left or on the right of the entailment symbol ($\vdash_{\mathscr{P}}$) in the consequent. A right introduction rule concludes a property of the constructed term, while a left introduction rule supports a conclusion drawn from assumptions about the specified term construction. Left introduction rules in sequent calculus are used to draw inferences similar to those made with so-called *elimination* rules of a natural deduction style logic.

5.3.1 Abstraction and function application

A predicate $P \to Q$ is satisfied by a function-typed term whose application to an argument with property P gives a result with property Q. The following rule asserts

[6] We say that the denotation of an expression is *normal* if it is not the bottom element in its type.

an arrow property of a Haskell term formed by explicit abstraction:

$$\frac{\Gamma[x :: \tau_1];\, \Pi,\, x ::: P \vdash_{\mathscr{P}} e :: \tau_2 ::: Q}{\Gamma;\, \Pi \vdash_{\mathscr{P}} (\lambda x \text{ -> } e) :: (\tau_1 \rightarrow \tau_2) \ ::: \ \$(P \rightarrow Q)} \tag{8}$$

The typing of terms in this rule has been shown explicitly.

As an illustration, we might apply Rule (8) to verify a property of an abstraction that gives a successor function at type *Integer*. An instance of the rule (in informal notation) would be

$$\frac{x :: \mathit{Integer};\ x \geqslant 0 \vdash_{\mathscr{P}} \ (1 + x) > 0}{\vdash_{\mathscr{P}} \ (\lambda(x :: \mathit{Integer}) \text{ -> } 1 + x) ::: \$(\,!(\geqslant 0) \rightarrow !(> 0))}$$

where $!(> 0)$ denotes a *right section* predicate constructed from the binary inequality operator, $(>)$. The antecedent clause in this example is a verification condition that might be discharged by applying a rule that expresses a property of integer arithmetic. We have not stated any such theory-specific rules in this paper.

Rule (8) accommodates the strictness properties of abstractions as they are defined in Haskell. An unstrengthened domain predicate, P, does not assert that the argument of an abstraction has a normal value. A property $\$(P \rightarrow Q)$ may therefore be satisfied by an abstraction that is not strict in its argument. To express a stronger property, one appropriate to an abstraction whose body is strict in the abstracted variable, we could assume an explicitly strengthened domain predicate, $\$P'$. In that case, the consequent property of the strict abstraction would become $\$(\$P' \rightarrow Q)$. If, in addition, we wanted to assert that the function defined by the abstraction is total when applied to an argument satisfying $\$P'$, the codomain predicate in the arrow property would also be strengthened, as in $\$(\$P' \rightarrow \$Q')$.

A rule for left introduction of an arrow property of a Haskell term is:

$$\frac{\Pi \vdash_{\mathscr{P}} e' ::: P \qquad e\, e' ::: Q \vdash_{\mathscr{P}} \Delta}{\Pi,\, e ::: \$(P \rightarrow Q) \vdash_{\mathscr{P}} \Delta} \tag{9}$$

Notice that the assumption in the consequent clause cannot be weakened to $e ::: P \rightarrow Q$, as the rule would then be unsound in the case that Q was substituted by a strengthened predicate.

5.3.2 *Application*

Rule (10) is a right-introduction rule for properties of function application. Notice that the conclusion of a strong arrow property for e_1 in the first antecedent is necessary. If the arrow property had only been weakly asserted in the conclusion of the first antecedent, then the rule would not be valid if the predicate variable Q was substituted by a strong property.

$$\frac{\Pi \vdash_{\mathscr{P}} e_1 ::: \$(P \rightarrow Q) \qquad \Pi \vdash_{\mathscr{P}} e_2 ::: P}{\Pi \vdash_{\mathscr{P}} e_1\, e_2 \ ::: \ Q} \tag{10}$$

A left introduction rule for application is:

$$\frac{e ::: P \rightarrow \$Q \vdash_{\mathscr{P}} \Delta}{x ::: P,\, e\, x ::: \$Q \vdash_{\mathscr{P}} \Delta} \tag{11}$$

where x is a term variable having no free occurrence in e. In this rule, which is dual to Rule (8), the restriction of the argument term, x, to a variable ensures that the property assumed of the application in the consequent is valid for any argument that satisfies the domain predicate, P.

5.3.3 *Tuple predicates*

A tuple predicate characterizes a property of a Haskell tuple expression. A right introduction rule allows one to conclude such a property:

$$\frac{\Pi \vdash_{\mathscr{P}} e_1 ::: P_1 \quad \cdots \quad \Pi \vdash_{\mathscr{P}} e_k ::: P_k}{\Pi \vdash_{\mathscr{P}} (e_1, \ldots, e_k) ::: \$(P_1, \ldots, P_k)} \tag{12}$$

A dual rule for left introduction applies when a property of a tupled expression is assumed:

$$\frac{\Pi,\, e_1 ::: P_1, \cdots, e_k ::: P_k \vdash_{\mathscr{P}} \Delta}{\Pi,\, (e_1, \ldots, e_k) ::: (P_1, \ldots, P_k) \vdash_{\mathscr{P}} \Delta} \tag{13}$$

5.3.4 *Negated predicates*

P-logic is a classical logic; however, the strength modality must be accounted for in formulating closure axioms for predicate negation. Predicate negation is an operation defined in terms of propositional negation of an assertion of a strengthened predicate,

$$x ::: \neg P \Leftrightarrow \neg(x ::: \$P)$$

and therefore, the proposition $\perp ::: \neg P$ is true at every type.

Closure axioms for negated predicates are:

$$\frac{}{\Gamma \vdash_{\mathscr{P}} x ::: \$P,\ x ::: \neg P} \tag{14}$$

$$\frac{}{x ::: \$P,\ x ::: \neg P \vdash_{\mathscr{P}} \Delta} \tag{15}$$

5.3.5 *Constructor application*

Rules for constructor application are derived from a Haskell data type declaration. A constructor application is lifted to a predicate constructor application by the function *conPred*, given in Figure 4, where *ts* is a list of strictness-type pairs. Each listed pair gives the sat-strictness of the constructor (either *Lazy* or *Strict*) and the type expected in the corresponding argument position. When a predicate constructor lifted from a data constructor is applied to a predicate argument, the resulting predicate is strong if and only if at every argument position declared sat-strict for the data constructor, a strong argument predicate is given. If the declaration of the data constructor did not specify sat-strictness in any argument position, then by default the lifted predicate is strong. A strong predicate formula, $\$C\,P_1 \ldots P_k$, where C is a data constructor of arity k, is satisfied by a term with a normal form $C\ e_1\ \ldots\ e_k$ if each of the e_j satisfies the corresponding predicate P_j.

```
conPred                      ::  E → [Pr] → Pr
conPred (Constr n ts) prs    =
      let  prs'   =  take (length ts) prs
           s      =  and (map (\(pr,l) → isStrong pr || l == Lazy)
                               (zip prs' (map fst ts)))
                        where  isStrong (Strong _)   =  True
                               isStrong _            =  False
      in   if s then Strong (ConPred n prs')
                else ConPred n prs'
```

Fig. 4. Lifting constructor applications to predicates.

Rule schemes for properties of saturated applications of data constructors are given below. Suppose *Constr C* $[(s_1,\sigma_1),\ldots,(s_k,\sigma_k)] \in \Sigma_{T\,\alpha_1\ldots\alpha_n}$. A rule scheme that specifies properties of expressions constructed with C is:

$$\frac{\Pi \vdash_{\mathscr{P}} e_1 ::: P_1 \;\cdots\; \Pi \vdash_{\mathscr{P}} e_k ::: P_k}{\Pi \vdash_{\mathscr{P}} C\, e_1 \ldots e_k ::: conPred\ (Constr\ C\ [(s_1,\sigma_1),\ldots,(s_k,\sigma_k)])\ [P_1 \ldots P_k]} \qquad (16)$$

Here, $0 \leqslant k$ and the predicate form in the consequent of the rule is given by an application of the function, *conPred*, which is defined in Figure 4. The first argument in this application is expressed in the abstract syntax representation of a constructor because the surface syntax does not carry the sat-strictness and arity attributes of the constructor that are extracted from its declaration. Although we have tried to present rules informally in terms of the surface syntax of terms and predicates whenever possible, the formal expression of rule (16) requires abstract syntax.

Notice from its definition that *conPred* calculates whether a constructed property is or is not strong. Its strength depends upon the sat-strictness attributes declared for a data constructor, C, and whether properties of its non-sat-strict arguments are asserted strongly.

A second rule satisfied by terms constructed with C is that for each data constructor, B, which is distinct from C in the same data type,

$$\frac{}{\Pi \vdash_{\mathscr{P}} C\, e_1 \ldots e_k ::: \neg B\ \underbrace{\mathsf{Univ}\ldots\mathsf{Univ}}_{arity\ of\ B}} \qquad (17)$$

Rule (17) asserts that terms constructed with different data constructors are semantically distinct.

There is a dual to rule (16) that expresses properties entailed by an assumed property of a constructed term. As before, assume C to be a k-place data constructor. Then,

$$\frac{\Pi,\, e_1 ::: P_1 \;\cdots\; e_k ::: P_k \vdash_{\mathscr{P}} \Delta}{\Pi,\, C\, e_1 \ldots e_k ::: \$C\ P_1 \cdots P_k \vdash_{\mathscr{P}} \Delta} \quad (0 \leqslant k) \qquad (18)$$

This rule tells us that any conclusion supported by properties assumed of terms $e_1,\ldots,e_k$ is also supported by assuming the constructor property of the constructed term, $C\ e_1 \cdots e_k\ :::\ \$C\ P_1 \cdots P_k$. In fact, the assumption made in the consequent of rule (18) is stronger than the conjoined assumptions given in the antecedent.

Consider the circumstance that the constructor, C, is sat-strict in its i^{th} argument, but the corresponding argument property, $e_i ::: P_i$, is only weakly asserted in the assumptions of the antecedent.

Rules (16) and (18) together reflect the embedding-projection property of data constructors expressed by equations (3)–(4).

5.4 Pattern matching

Pattern-matching, as a language feature, has the attractive aspect that it offers an intuitive interpretation of its surface syntax. However, formal reasoning about patterns is complicated by the fact that control and binding aspects occur together, and binding may encompass several variables at once. This section develops algorithms for deriving predicates from Haskell patterns. The derivation associates predicate arguments with the variables that occur in a pattern, so that a derived pattern predicate characterizes both the control aspect of a pattern and required properties of subterms of a matching term.

5.4.1 Pattern predicates

Because patterns may be nested to arbitrary depths, it is inconvenient to use the syntax of patterns directly in formulating proof rules. Instead, we shall define an algorithmic calculation of a syntactically flattened representation for patterns to support a presentation of pattern predicates in rule schemes. This will make it easier to account for predicate components associated with particular pattern variables bound in a nested pattern.

Definition 15 (*Pattern predicate*)
The *pattern predicate* formed by instantiating a pattern relative to a predicate environment is calculated by the inductively-defined Haskell function *pi*[7] given in Figure 5. We use the notation $\pi(p)$ in Rules (19)–(22) as shorthand for *pi p* to designate a “flattened” pattern predicate constructor. A rule scheme specified with π can be directly implemented as a rule generator, yielding a distinct rule for each instance of a pattern or patterns in terms to which it is applied.

Intuitively, π is a function that interprets an abstract syntax term that represents a pattern, substituting a predicate for each binding occurrence of a variable in the pattern. The predicates to be substituted are drawn from a list given as the second argument to π. The calculation yields a new predicate, which we refer to as a pattern predicate. However, calculation of a pattern predicate from a pattern is not simply a matter of substituting predicates for term variables. To obtain a predicate that characterizes terms matching the pattern, it is also necessary to interpret irrefutable patterns and the strictness annotations embedded in the signatures of data constructors.

[7] To make legal Haskell of the definitions in Fig. 5, the *State* type should be declared a **newtype** with a redundant data constructor. We have omitted the data constructor from the definitions given in the paper to economize on notational clutter.

```
  -- the pattern predicate
pi                          :: P → [Pred] → Pred
pi p predlist               = fst (patPred p predlist)
patPred                     :: P → [Pred] → (Pred, [Pred])
patPred (Pvar x) preds      = (pred, preds)
patPred Pwildcard preds     = (Univ, preds)
patPred (Ptilde p) preds    = let
                                 l             = length (fringe p)
                                 prs           = take l preds
                                 isUniv Univ   = True
                                 isUniv _      = False
                               in
                               if and(map isUniv prs) then
                                 (Univ, drop l preds)
                               else
                                 patPred p preds
patPred (Ptuple pats) preds = let
                                   (prs, preds') = mapS patPred pats preds
                                 in
                                 (Strong (TuplePred prs), preds')
patPred (Pcondata n lspats) preds =
    let
          (ss, pats)           = unzip lspats
          (prs, preds')        = mapS patPred pats preds
          ifStrict _ (Strong p) = Strong p
          ifStrict Strict p    = Strong p
          ifStrict Lazy p      = p
    in
        (Strong (ConPred n (zipWith ifStrict ss prs)), preds')

mapS f [] s        = ([], s)
mapS f (x : xs) s  = let (r, s') = f x s in mapS f xs s'

fringe                  :: P → [Name]
fringe (Pvar x)         = [x]
fringe (Ptuple ps)      = concat (map fringe ps)
fringe (Pcondata _ ps)  = concat (map (fringe . snd) ps)
fringe Pwildcard        = []
fringe (Ptilde p)       = fringe p
```

Fig. 5. Calculation of pattern predicates. The functions *fst*, *snd*, *take*, *drop* and *zipWith* are defined in the Haskell standard prelude.

When an irrefutable pattern occurs as the first argument of π and every member of the list of predicates that would replace variables in the pattern's *fringe* (see Defn. 16) is Univ, the pattern predicate returned is Univ, regardless of the substructure of the irrefutable pattern. Otherwise, the "skeleton" of each subpattern is fully elaborated by $\pi(p)$. In consequence, if an instance of rule (19) has non-universal predicates among its hypotheses, then the pattern predicate in its conclusion will characterize a normal pattern match.

As an illustration, three pattern predicates that may be calculated from the patterns given as examples of Section 2.2.6 are shown below. For easier readability, the patterns and the resulting pattern predicates are shown in concrete, rather than abstract syntax representations.

$$\begin{aligned}
\pi(\texttt{T (S x) y})\ \mathsf{Univ}\ \mathsf{Univ} &= \$(\texttt{T}\ \$(\texttt{S}\ \mathsf{Univ})\ \mathsf{Univ}) \\
\pi(\texttt{T} \sim\texttt{(S x) y})\ \mathsf{Univ}\ \mathsf{Univ} &= \$(\texttt{T}\ \mathsf{Univ}\ \mathsf{Univ}) \\
\pi(\texttt{T} \sim\texttt{(S x) y})\ \$\mathsf{Univ}\ \mathsf{Univ} &= \$(\texttt{T}\ \$(\texttt{S}\ \$\mathsf{Univ})\ \mathsf{Univ})
\end{aligned}$$

Let us focus attention on the predicates given as arguments to the pattern constructor in the left-hand side of each of the equations above. In the first equation, both argument predicates are Univ, which is satisfied by any term (including the term *undefined*) that might be bound to the variables x and y in a pattern match. Nevertheless, the fact that the sub-pattern S x has a data constructor at its head mandates that in any term on which a match is to succeed, the first argument of the data constructor T must have a normal value. Hence, the pattern predicate embeds a strong pattern as the first argument of the (lifted) constructor, T.

Although the argument predicates are the same in the second equation as in the first, (~) at the front of the sub-pattern indicates that matching of this sub-pattern will not be effective unless a value is demanded for the variable, x. Since demand for a variable cannot be determined from a pattern (it depends upon the evaluation context), the pattern predicate in the first argument position of the constructor, T is Univ. The predicate derived from the pattern cannot be made more precise.

In the third equation, the strengthed predicate $Univ is asserted of a term bound to the variable x in a pattern match. This asserts that any value bound to x must be non-bottom. Consequently, the second argument of the constructor T in the pattern predicate is asserted to have a normal value matching S x, in spite of the (~) prefix of the pattern. The assertion that x has a strong property is, in essence, an assertion that an actual value for x might be demanded in an evaluation context.

Definition 16
[Fringe of a pattern] The *fringe* of a pattern p is the list of (distinct) variables occurring in p, in left-to-right order. It is formally defined on the abstract syntax of patterns by the Haskell function *fringe* given in Fig. 5.

The fringe of a pattern p is closely related to the record type of its codomain, as defined in Definition 8. Specifically, if $p :: \tau \rightarrow \zeta$ and $\zeta = \{x_1 : \sigma_1, \ldots, x_n : \sigma_n\}$, then $fringe(p)$ is a list (without repetitions) of the variables $x_1, ..., x_n$, arranged in order of their left-to-right occurrence within p.

5.4.2 The domain of a pattern

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

Definition 17
[Pattern Domain Predicate] The *domain predicate* of pattern p, called $Dom(p)$, is the predicate defined by applying the predicate pattern constructor derived from p to a list of Univ predicates.

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

Notice that $Dom(p)$ is either Univ (in case the pattern is a variable, is the wildcard pattern, or is irrefutable) or it is a strong predicate.

The formula $\neg Dom(p)$ asserts that a term fails to match p or is undefined. Thus, a strengthened domain predicate disjoined with its strong complement is, in effect, a partial definedness predicate. A term that satisfies either $\$Dom(p)$ or $\$\neg Dom(p)$ must have a normal value at every subterm necessary to evaluate a control-enabled match with the pattern p.

5.4.3 *Properties of case branches*

There are two rule schemes for case branches. We write a case branch as $\{p \rightarrow e\}$, where the meta-variable p represents the pattern, and e the expression in a case branch. One rule characterizes the function of a case branch when it is tried in a case expression whose discriminator matches its pattern:

$$\frac{\Pi, x_1 ::: P_1, \cdots, x_n ::: P_n \vdash_{\mathscr{P}} e ::: Q}{\Pi \vdash_{\mathscr{P}} \{p \texttt{->} e\} ::: \pi(p)\,P_1 \cdots P_n \rightarrow \$\texttt{Just}\ Q} \tag{19}$$

where $[x_1, \ldots, x_n] = \mathit{fringe}\ p$, and a second rule characterizes its behavior when pattern-matching fails:

$$\Pi \vdash_{\mathscr{P}} \{p \texttt{->} e\} ::: \$\neg Dom(p) \rightarrow \$\texttt{Nothing} \tag{20}$$

Individual branches of a case expression are logically characterized by arrow predicates, where data constructors of the *Maybe* data type occur in the codomain predicate to code the success or failure of a match on the pattern of a case branch.

5.4.4 *Properties of case expressions*

Recall from Section 5.4 that predicates associated with the case branches of a case expression have the form *Just P*, for some predicate P, or else *Nothing*. We refer to such predicates as *Maybe* predicates. The intuition behind the *Maybe* predicate, *Nothing*, is that no specific property can be inferred from it. It characterizes a case expression whose result is undefined, or whose attempted evaluation results in a run-time error.

Rules for a case expression are defined inductively, based upon a rule for a single case branch. The base for induction is given in terms of a pseudo case expression in the rules below. The keyword `caseM` does not actually belong to the Haskell language, but is used in these rules to designate a pseudo-expression form whose properties are expressed by *Maybe* predicates.

$$\frac{\Pi \vdash_{\mathscr{P}} d ::: \pi(p)\,P_1 \cdots P_k \qquad \Pi \vdash_{\mathscr{P}} \mathit{match} ::: \pi(p)\,P_1 \cdots P_k \rightarrow \$\texttt{Just}\ Q}{\Pi \vdash_{\mathscr{P}} \texttt{caseM}\ d\ \texttt{of}\ \{\mathit{match}\} ::: \$\texttt{Just}\ Q} \tag{21}$$

$$\frac{\Pi \vdash_{\mathscr{P}} d ::: \$\neg Dom(p)}{\Pi \vdash_{\mathscr{P}} \texttt{caseM}\ d\ \texttt{of}\ \{p\ \texttt{->}\ e\} ::: \$\texttt{Nothing}} \tag{22}$$

Notice that for an irrefutable pattern, $Dom({\sim}p') = \mathsf{Univ}$ and thus, $\$\neg Dom({\sim}p') = \$\mathsf{UnDef}$, which is unsatisfiable. Thus the antecedent of rule (22) cannot be discharged when p is an irrefutable pattern, as it might be if p were an ordinary constructor pattern.

The following rules account for a Haskell case expression, without guards[8].

$$\frac{\Pi \vdash_{\mathscr{P}} \texttt{caseM}\ d\ \texttt{of}\ \{match\} ::: \$\texttt{Nothing} \qquad \Pi \vdash_{\mathscr{P}} \texttt{case}\ d\ \texttt{of}\ \{matches\} ::: Q}{\Pi \vdash_{\mathscr{P}} \texttt{case}\ d\ \texttt{of}\ \{match;\ matches\}\ ::: Q} \tag{23}$$

$$\frac{\Pi \vdash_{\mathscr{P}} \texttt{caseM}\ d\ \texttt{of}\ \{match\}\ :::\ \$\texttt{Just}\ P}{\Pi \vdash_{\mathscr{P}} \texttt{case}\ d\ \texttt{of}\ \{match;\ matches\}\ :::\ P} \tag{24}$$

where *matches* is a sequence of zero or more case branches.

5.5 Using P-logic

In this section, we provide three concrete examples illustrating the use of P-logic in the specification and verification of demand-oriented Haskell programs. The first two such examples, presented in section 5.5.1, demonstrate how pattern-match success and failure are manifested in the logic. Section 5.5.2 presents a second, extended example concerning the verification of a property of the client-server model presented in the on-line Haskell tutorial (Hudak *et al.*, 2000). This example illustrates the formulation of a useful property along with its verification; the verification highlights the calculation of pattern predicates and their rôle in the inference rules of P-logic. These examples demonstrate how P-logic explains complicated demand-related behavior in Haskell.

5.5.1 Example: pattern-matching success and failure in P-logic

Below are two sample derivations demonstrating how P-logic distinguishes pattern-matching success and failure. The first derives a strong property of a case expression in which there is a pattern matching the case discriminator.

$$\dfrac{\dfrac{\dfrac{}{\vdash \mathtt{L} ::: \mathsf{Univ}} \qquad \dfrac{}{\vdash \mathtt{R} ::: \$\mathsf{R}}\,(16)}{\vdash (\mathtt{T\,L\,R}) ::: \$(\mathsf{T\,Univ\,\$R})}\,(16) \qquad \dfrac{\dfrac{}{\mathtt{x} ::: \mathsf{Univ}, \mathtt{y} ::: \$\mathsf{R} \vdash \mathtt{y} ::: \$\mathsf{R}}}{\vdash \{(\mathtt{T\tilde{}(S\,x)\,y)\ \texttt{->}\ y}\} ::: \$(\mathsf{T\,Univ\,\$R}) \to \$\mathsf{Just}\ \$\mathsf{R}}\,(19)}{\dfrac{\vdash \texttt{caseM}\ (\mathtt{T\,L\,R})\ \texttt{of}\ \{\ (\mathtt{T\tilde{}(S\,x)\,y)\ \texttt{->}\ y}\ \} ::: \$\mathsf{Just}\ \$\mathsf{R}}{\vdash \texttt{case}\ (\mathtt{T\,L\,R})\ \texttt{of}\ \{\ (\mathtt{T\tilde{}(S\,x)\,y)\ \texttt{->}\ y}\ \} ::: \$\mathsf{R}}\,(24)}\,(21)$$

The pattern property at the application of rule (21) above, which is calculated by the function *patPred* given in Fig. 5, is

$$\pi(\mathtt{T\ \tilde{}(S\ x)\ \ y})\ \mathsf{Univ}\ \$\mathsf{R} = \$(\mathsf{T\,Univ\,\$R})$$

[8] It is straightforward to extend *P*-logic to account for case branches with guards, by using *Maybe* predicates. Guards have not been included in the Haskell fragment on which this paper is based because they add nothing essential to the exposition.

A second derivation involves a `case` branch that generates a pattern match failure. The conclusion of such a derivation provides one of the antecedents needed for an application of rule (23), which accounts for a case expression in which failure of a pattern match causes a branch to be bypassed.

$$\dfrac{\dfrac{}{\vdash \texttt{(T L R)} ::: \$\neg(\mathsf{S\ Univ})}\ (17)}{\vdash \texttt{caseM (T L R) of } \{\texttt{(S x) -> x}\} ::: \$\texttt{Nothing}}\ (22)$$

It is important to note how provability in P-logic explains pattern-match failure – there is no way of "getting rid of" the `caseM` construct in the conclusion. It is clear from Rules (23) and (24) that, if the above branch is not part of some larger case expression, then nothing further may be derived about this match failure.

5.5.2 *Example: the client-server model*

The example of section 5.5.1 relies upon a pattern predicate for the case match, but the calculation of this predicate was not highlighted in either sequent proof presented there. The next example illustrates a pattern predicate calculation.

The on-line Haskell tutorial (Hudak *et al.*, 2000) illustrates deferred pattern-matching with a simple client-server model. The interaction of client and server processes is modeled by streams of requests issued by the client with responses returned by the server. An irrefutable pattern in the definition of the `client` function specifies deferred pattern matching, which makes the mutually-recursive definitions of request and response streams well-founded.

```
reqs  = client init resps
resps = server reqs
  where
    client  = λ init resps ->
                    case resps of
                       ~(resp : resps') -> init  : client (next resp) resps'
    server  = λ reqs ->
                    case reqs of
                       (req : reqs') -> process req  : server reqs'
```

For simplicity, the requests and responses are both modeled as *Integer*-typed values generated by the primitives

```
init         = 0
next resp    = resp
process req  = req + 1
```

The definitions of `client` and `server` given in the tutorial have been "desugared" here in order to put their right-hand sides into the form of an abstraction over a `case` expression. Most Haskell compilers perform syntactic desugaring to reduce the number of distinct abstract syntax constructors that must be analyzed, and a theorem proving assistant can be expected to provide this service as well.

A client process is expected to start the interaction by issuing an initial request – `init` in the example. However, the function declared above as `client` will return the construction of a list only if the pattern-match on its second argument succeeds. When the client and server processes are first started, no computation has yet been done, and the list of responses that `client` expects as its second argument has not yet been constructed. The recursion scheme shows that the list of responses will not be constructed until the (initial) construction of a list of requests is available for pattern-matching by the `server` function. Thus the pattern given in the definition of `client` has been declared irrefutable, to allow an actual match to be deferred. Had this pattern not been given the (~) annotation, the recursive definition of `reqs` and `resps` would be unfounded, and an expression calling for an initial prefix of either of these lists would fail to evaluate.

The crucial property (well-foundedness) is easy to formalize in *P*-logic with the assertion of the strong property that `reqs` evaluates to a constructed list:

$$\texttt{reqs} ::: \$(\mathsf{Univ} : \mathsf{Univ})$$

Let's examine a few critical steps in a proof of this assertion. The definitions of constants `client`, `server`, `reqs` and `resps`, taken directly from the program text, are assumed as equalities in the logical context, Π, of a sequent representing the assertion. In addition, Π may contain trivial assumptions of the form e ::: Univ, where e is any Haskell expression. Usually, we take trivial assumptions to be implicit.

After using the assumed equalities to replace occurrences of `reqs` and subsequently, `client` by their definitions in the asserted conclusion and applying rules (10) and (8), the verification condition for the sequent representing the assertion becomes:

$$\Pi \vdash_{\mathscr{P}} \{\texttt{case}\ x_1\ \texttt{of}\ \sim\!(x_2 : x_3)\texttt{->}\ x_0\ :\ \texttt{client}\ (\texttt{next}\ x_2)\ x_3\} ::: \$(\mathsf{Univ} : \mathsf{Univ}) \tag{25}$$

where the variables bound in the definition of `client` and its `case` match have been renamed to fresh variables x_0, x_1, x_2 and x_3 by alpha-conversion.

Further application of rules (24) and (21) yields the following verification condition for the case branch,

$$\Pi \vdash_{\mathscr{P}} \{\sim\!(x_2 : x_3)\texttt{->}\ x_0\ :\ \texttt{client}\ (\texttt{next}\ x_2)\ x_3\} ::: \mathsf{Univ} \rightarrow \$\texttt{Just}\,(\$(\mathsf{Univ} : \mathsf{Univ})) \tag{26}$$

As no nontrivial property of the case discriminator is assumed, the predicate to the left of the arrow constructor in the verification condition above is Univ, which is the trivial property implicitly assumed of the discriminator. A necessary condition to discharge (26) is that under the (trivial) assumptions x_2 ::: Univ and x_3 ::: Univ, the pattern property calculated of $\sim\!(x_2 : x_3)$ is implied by Univ. This pattern property is computed by:

$$\mathit{patPred}\ (\mathit{Ptilde}\,(\mathit{Pcondata}\ \text{“:”}\ [(\mathit{Lazy}, \mathit{Pvar}\ x_2), (\mathit{Lazy}, \mathit{Pvar}\ x_3)]))\ [\mathsf{Univ}, \mathsf{Univ}] \tag{27}$$

Note that, using the rules in Figure 5, the above pattern property does indeed reduce to Univ, as every member of the list of predicate arguments is Univ and the Haskell list constructor is lazy in both its arguments.

It is important to note how P-logic explains the use of ~ here. Had the pattern not been annotated with ~, the computation of

$$\mathit{patPred}\ (\mathit{Pcondata}\ \text{“:”}\ [(\mathit{Lazy}, \mathit{Pvar}\ x_2), (\mathit{Lazy}, \mathit{Pvar}\ x_3)])\ [\mathsf{Univ}, \mathsf{Univ}]$$

would have yielded $(Univ : Univ) as the predicate appearing to the left of the arrow constructor in sequent (26). Since this property is stronger than can be assumed of an unevaluated case discriminator (x_1 in sequent (25) above), an attempted proof of the strong property of **reqs** would fail.

Since the pattern predicate in (27) conforms to that which is expected, rule (19) applies to sequent (26), yielding as the final verification condition,

$$\Pi \vdash_{\mathscr{P}} x_0 \ : \texttt{client}\ (\texttt{next}\ x_2)\ x_3 ::: \$(\mathsf{Univ} : \mathsf{Univ}) \tag{28}$$

The context, Π, implicitly contains x_0 ::: Univ and `client` (`next` x_2) x_3 ::: Univ, thus we are left to check the logical congruence induced by the list constructor ":". This data constructor inherits from its declaration in Haskell's standard prelude the abstract syntax representation *Constr* ":" $[(\mathit{Lazy}, a), (\mathit{Lazy}, [a])]$. The logical congruence, calculated by the function *conPred* given in Figure 4, yields

$$\begin{aligned} &\mathit{conPred}\ (\mathit{Constr}\ \text{“:”}\ [(\mathit{Lazy}, a), (\mathit{Lazy}, [a])])\ [\mathsf{Univ}, \mathsf{Univ}] \\ &\quad = \quad \mathit{Strong}\ (\mathit{ConPred}\ \text{“:”}\ [\mathsf{Univ}, \mathsf{Univ}]) \end{aligned}$$

which in concrete syntax, is the asserted property $(Univ : Univ), discharging (28).

5.6 A semantic interpretation of P-logic

A model for *P*-logic extends a Haskell frame model by providing interpretations for predicate constants and predicate constructors. The meanings of predicates refine the meanings of types. The meaning of a simply typed predicate in *P*-logic is defined as a characteristic predicate over the set underlying a frame that interprets the corresponding Haskell term type, τ.

Let $\mathscr{D}[\![\_]\!]_\tau :: \mathit{Term} \to \mathit{Env} \to |D_\tau|$ be a meaning function that maps every τ-typed Haskell expression to its denotation in the underlying set of a frame object, $\mathscr{D}_\tau$, where $\mathit{Env} = \mathit{Var} \to |D|$. When the model is evident from context, as when we are only talking about a single model, the model identifier will be omitted from the meaning function.

We shall overload the meaning-brackets notation to express the semantics of predicate formulas at a type, τ, $[\![-]\!]_\tau :: \mathit{Predicate} \to \mathit{PredEnv} \to \mathit{Powerset}\,|D_\tau|$, where *PredEnv* is the type of a predicate environment that gives meanings to predicate variables. We need predicate environments because the rules of *P*-logic contain predicate variables that range over formulas.

Definition 18
A *predicate assignment*, ξ, is a type-indexed set of maps from predicate identifiers to sets of denotations in the type given by the index. The type of a predicate assignment is $\mathit{PredEnv} :: \bigcup_{\tau \in \mathsf{Type}} \{\mathit{Name} \to \mathit{Powerset}\,|D_\tau|\}$. A predicate assignment gives meanings to predicate variables in its domain at every type.

5.6.1 Strong predicates

Formulas are interpreted as characteristic predicates of sets (posets) in a type frame. Given that the meaning of a predicate formula P of type $\mathit{Pred}\ \tau$ is a subset of the τ-type frame, $[\![P]\!]_\tau\,\xi \subseteq |D_\tau|$, the interpretation of a strong predicate is

$$[\![\mathit{Strong}\ P]\!]_\tau\,\xi = [\![P]\!]_\tau\,\xi \setminus \{\bot_\tau\}$$

5.6.2 Universal predicates

The predicate constants Univ and UnDef represent the universal predicate and the predicate satisfied only by the bottom element, in each type frame. The interpretations of these predicates are:

$$[\![\mathsf{Univ}]\!]_\tau\,\xi = |D_\tau| \qquad [\![\mathsf{UnDef}]\!]_\tau\,\xi = \{\bot_\tau\}$$
$$[\![\mathsf{\$Univ}]\!]_\tau\,\xi = |D_\tau| \setminus \{\bot_\tau\} \qquad [\![\mathsf{\$UnDef}]\!]_\tau\,\xi = \{\,\}$$

5.6.3 Predicate variables

The meaning assigned to a predicate variable at a specified type is given by applying the predicate environment map at that type to the name of the variable:

$$[\![PredVar\ n]\!]_\tau\,\xi = \xi_\tau\,n$$

5.6.4 Data-induced congruence predicates

The meaning of a predicate formed with a k-ary data constructor, C, at a ground instance of a Haskell data type, T, is given by the following:

$$\begin{array}{l}
\text{If } Constr\ C\ [(s_1,\tau_1),\ldots,(s_k,\tau_k)] \in \Sigma_T \text{ then} \\
[\![Conpred\ C\ [P_1 \cdots P_k]]\!]_T\,\xi = \\
\quad \{c \bullet (t_1,\ldots,t_k) \mid t_1 \in [\![P'_1]\!]_{\tau_1}\,\xi \wedge \ldots \wedge t_k \in [\![P'_k]\!]_{\tau_k}\,\xi\} \cup \{\bot\} \\
\text{where } P'_i = \begin{cases} \$P_i & \text{if } s_i = Strict \\ P_i & \text{if } s_i = Lazy \end{cases} \\
\text{and } c^\sharp \in |D_{\tau_1 \to \cdots \to \tau_k \to T}| \text{ is the semantic embedding of } C
\end{array}$$

5.6.5 Arrow predicates

An arrow predicate characterizes a property of a function-typed term. We can read a proposition such as $e ::: P \to Q$ as the assertion "when e is applied to an argument that has property P, the application has property Q".

$$\begin{array}{l}
[\![Parrow\ P\ Q]\!]_{\tau_1\to\tau_2}\,\xi = \\
\{f \in |D_{\tau_1\to\tau_2}| \mid \forall x.\, x \in [\![P]\!]_{\tau_1}\,\xi \Rightarrow f \bullet x \in [\![Q]\!]_{\tau_2}\,\xi\} \cup \{\bot_{(\tau_1\to\tau_2)}\}
\end{array}$$

where the function space is that of continuous functions from $|D_{\tau_1}|$ to $|D_{\tau_2}|$.

5.6.6 Negated predicates

The meaning of a negated predicate is the complement of the meaning of the positive predicate with respect to the frame set of its type, to which the bottom element of the type frame is appended.

$$[\![Pneg\ P]\!]_\tau\,\xi = (|D_\tau| \setminus [\![P]\!]_\tau\,\xi) \cup \{\bot_\tau\}$$

5.6.7 Polymorphic predicates

The meaning of a polymorphic predicate is not given directly. Rather, a polymorphically typed term is said to satisfy a compatibly typed predicate if and only if every ground-typed instance of the term satisfies the corresponding ground-typed instance of the predicate.

Definition 19
A well-typed predicate, P, is *polymorphic* in a type variable, α, if it has a typing $\Gamma \vdash P :: Pred\ \sigma$, where $\alpha \in FV(\sigma)$.

5.7 Satisfiability and validity of a sequent

This section will formalize the notion of what it means for a well-typed term to satisfy a compatibly typed predicate, stating it in the setting of type frame semantics.

Definition 20
[Ground proposition]
Let $\mathscr{A}$ be a ground type environment and $\tau \in \mathsf{Type}$. If a term e and predicate symbol P satisfy the typing judgments $\mathscr{A} \vdash e::\tau$ and $\vdash P::Pred\,\tau$, where τ is the (ground) type derived for e in $\mathscr{A}$, then, $\mathscr{A} \vdash e::\tau\ :::\ P$ is a *ground proposition* in $\mathscr{A}$. A set of propositions, Π, is *ground in* $\mathscr{A}$ (which we write as $\mathscr{A} \vdash \Pi$) if every $\pi \in \Pi$ is a ground proposition in $\mathscr{A}$.

Definition 21
[Truth of a ground proposition in a frame model]
Let $\mathscr{D}$ be a Haskell frame as defined in section 4 and let $\mathscr{A}$ be a ground type environment. Suppose term e and predicate symbol P satisfy the typing judgments $\mathscr{A} \vdash e::\tau$ and $\vdash P::Pred\,\tau$, respectively. Further, let ρ be an $\mathscr{A}$-compatible value assignment and ξ be a predicate assignment. We say that the ground proposition $\mathscr{A} \vdash e::\tau\ :::\ P$ is ***true*** in frame $\mathscr{D}$ under assignments ρ and ξ iff $\mathscr{D}[\![\mathscr{A} \vdash e::\tau]\!]\rho \in \mathscr{D}[\![P]\!]_\tau\,\xi$. We write $\mathscr{A};\mathscr{D},\rho,\xi \models Pr$ to express that a proposition Pr, well-typed in $\mathscr{A}$, is true in a specific frame model and environment.

Definition 22
[Ground sequent]
Let $\mathscr{A}$ be a ground type environment. A sequent $\mathscr{A};\Pi \vdash_{\mathscr{P}} \Delta$ is ***ground in*** $\mathscr{A}$ if both $\mathscr{A} \vdash \Pi$ and $\mathscr{A} \vdash \Delta$.

Definition 23
[Polymorphic sequent]
Let Γ be a type environment containing free occurrences of type variables. A sequent $\Gamma;\Pi \vdash_{\mathscr{P}} \Delta$ is ***polymorphic in*** $FV(\Gamma)$ if for all $\mathscr{A}$ in TA(Γ), the sequent $\mathscr{A};\Pi \vdash_{\mathscr{P}} \Delta$ is ground in $\mathscr{A}$.

Definition 24
[Validity of a ground sequent]
Let $\mathscr{D}$ be a Haskell frame and $\mathscr{A}$ a ground type environment. A ground sequent $\mathscr{A};\Pi \vdash_{\mathscr{P}} \Delta$ is ***valid for*** $\mathscr{D}$ ***under predicate assignment*** ξ if, for every $\mathscr{A}$-compatible value assignment, ρ, the following implication is true:

$$(\forall Pr \in \Pi.\ \mathscr{A};\mathscr{D},\rho,\xi \models Pr) \ \Rightarrow\ \exists Pr' \in \Delta.\ \mathscr{A};\mathscr{D},\rho,\xi \models Pr'$$

Definition 25

[Validity of a polymorphic sequent]
Let $\mathscr{D}$ be a Haskell frame and Γ be a non-ground type environment. A polymorphic sequent $\Gamma; \Pi \vdash_{\mathscr{P}} \Delta$ is ***valid for $\mathscr{D}$ under predicate assignment*** ξ if forall $\mathscr{A}$ in $\mathrm{TA}(\Gamma)$, $\mathscr{A}; \Pi \vdash_{\mathscr{P}} \Delta$ is valid for $\mathscr{D}$ under ξ. We write $\mathscr{D}, \xi \models \varphi$ to express that a polymorphic sequent, φ is valid for $\mathscr{D}$ under ξ.

5.7.1 *Satisfiability of polymorphic predicates*

The typing discipline ensures that the meaning of a predicate that is polymorphic in a type variable α cannot depend upon the structure of terms of type α. If a polymorphically typed expression is specialized by a value assignment to a (polymorphically typed) term variable and satisfies a predicate under a ground type assignment, $\mathscr{A}$, then it also satisfies the predicate when specialized by a value assignment under another type assignment, $\mathscr{A}'$. We formalize this assertion in the following lemma.

Some notation is introduced in the statement of the lemma. If e is a Haskell term, the restriction of ρ to free variables of e is expressed as $\rho \downarrow_{FV(e)}$. Also, let $+\!\!> :: (\mathit{Vars} \to D) \times (\mathit{Vars} \to D) \to (\mathit{Vars} \to D)$ be the environment-extending function specified by the equation $(\rho +\!\!> \rho')\ x = \textbf{if}\ x \in \mathsf{dom}(\rho')\ \textbf{then}\ \rho'\ x\ \textbf{else}\ \rho\ x$.

Lemma 1

[Polymorphic Predicates]
Let Γ be a typing environment, σ a type scheme and suppose $e :: \sigma$ is a term well-typed in Γ and $P :: \mathit{Pred}\ \sigma$ is a unary predicate.

$$\begin{array}{l}
\forall \mathscr{A}_1, \mathscr{A}_2 \in \mathrm{TA}(\Gamma). \\
\quad \exists! \theta_1.\ \mathscr{A}_1 = \theta_1^* \circ \Gamma \Rightarrow \\
\quad \exists! \theta_2.\ \mathscr{A}_2 = \theta_2^* \circ \Gamma \Rightarrow \\
\qquad \forall \rho, \rho_1, \rho_2 :: \mathit{Vars} \to D \setminus \{\bot\}. \\
\qquad\quad \forall \xi :: \mathit{PredEnv}. \\
\qquad\qquad \mathit{Dom}(\rho_1) = \mathit{Dom}(\rho_2) = \{x \in \mathit{Vars} \mid \mathrm{TV}(\Gamma\ x) \neq \emptyset\} \Rightarrow \\
\qquad\qquad\quad \Gamma \models \rho\downarrow_{FV(e)}\ \wedge \mathscr{A}_1 \models \rho_1\downarrow_{FV(e)}\ \wedge \mathscr{A}_2 \models \rho_2\downarrow_{FV(e)} \Rightarrow \\
\qquad\qquad\qquad [\![\mathscr{A}_1 \vdash e :: \theta_1^*\,\sigma]\!]\,(\rho +\!\!> \rho_1) \in [\![\vdash P]\!]_{\theta_1^*\,\sigma}\,\xi \\
\qquad\qquad\qquad\qquad \Longleftrightarrow \\
\qquad\qquad\qquad [\![\mathscr{A}_2 \vdash e :: \theta_2^*\,\sigma]\!]\,(\rho +\!\!> \rho_2) \in [\![\vdash P]\!]_{\theta_2^*\,\sigma}\,\xi
\end{array}$$

The lemma asserts that satisfaction of a strong predicate by a term in any type-respecting interpretation is independent of the value assignment made to polymorphically typed term variables. The polymorphic typing condition on a variable, x, is $\mathrm{TV}(\Gamma\ x) \neq \emptyset$. The type compatibility condition $\Gamma \models \rho\downarrow_{FV(e)}$ provides for variables that occur free in e but which are not polymorphically typed in Γ; any such variable will have a value assigned in ρ and this assignment must be compatible with the typing given by Γ. The restriction of value assignments ρ_1 and ρ_2 to non-bottom values eliminates the possibility that one of these assignments produces bottom while the other does not. As bottom is an element of every type, this restriction does not limit the scope of assigned values that might distinguish types.

Proof
We consider explicitly only atomic predicates; the proof extends to formulas constructed with predicate negation, conjunction and disjunction by an obvious induction. For atomic predicates we shall use coinduction on the structure of evaluation contexts that observe values manifesting the type scheme, σ.
Case $\sigma = \alpha$: If P is satisfiable at an arbitrary type instance, it must be that $P = \mathsf{Univ}$. Thus for any type instance $[\tau/\alpha]$ and any type-compatible valuation assignment ρ and predicate assignment ξ, $[\![\Gamma \vdash e :: \tau]\!]\,\rho \in [\![\Gamma \vdash \mathsf{Univ}]\!]_\tau\,\xi$, from which the conclusion of the lemma follows immediately.

Case $\sigma = T\ \alpha_1 \cdots \alpha_n = \cdots | C_j \sigma_{j,1} \ldots \sigma_{j,k_j} | \cdots$ where $j \in [1..m]$. If P is satisfiable, either $P = \mathsf{Univ}$ or P has the form $C_j\ P_{j,1} \cdots P_{j,k_j}$ for some $j \in [1..m]$. Consider the latter case. An expression $e :: \sigma$ is observed by a case expression. Individual components of a value constructed with a data constructor C_j are projected by expressions $\mathsf{case}\ e\ \mathsf{of}\ \{C_j\ x_1 \ldots x_{k_j} \to x_p\}$ for $p \in [1..k_j]$. As hypotheses for coinduction, assume the conclusion of the lemma for each of the typed assertions,

$$\Gamma \vdash \mathsf{case}\ e\ \mathsf{of}\ \{C_j\ x_1 \ldots x_{k_j} \to x_p\} :: \sigma_p ::: P_{j,p} \qquad (j \in [1..m],\ p \in [1..k_j])$$

As the assumed instances cover all projections from a term of the polymorphic data type, these hypotheses support the conclusion of the lemma for any well-typed proposition in the data type.

Case $\sigma = \sigma_1 \to \sigma_2$: If P is satisfiable, either $P = \mathsf{Univ}$ or P has the form $P_1 \to P_2$, where $P_1 :: \mathit{Pred}\ \sigma_1$ and $P_2 :: \mathit{Pred}\ \sigma_2$. The former case is immediate; so consider the latter. A value of an arrow type is observed by its applications to compatibly typed arguments. For any term, e', which satisfies the typing $\Gamma \vdash e' :: \sigma_1$, choose type environments $\mathscr{A}_1$ and $\mathscr{A}_2$ to instantiate the type scheme. Assume as hypotheses that the conclusion of the lemma holds (with the same choice of type environments, $\mathscr{A}_1$ and $\mathscr{A}_2$) for both the assertions $\Gamma \vdash e' :: \sigma_1 ::: P_1$ and $\Gamma \vdash e\ e' :: \sigma_2 ::: P_2$. Now, using the type frame equation at each instance of the polymorphic types gives

$$\begin{array}{l}
\forall \rho, \rho_1, \rho_2 :: \mathit{Vars} \to D\backslash\{\bot\}.\\
\quad \forall \xi :: \mathit{PredEnv}.\\
\quad\quad \mathit{Dom}(\rho_1) = \mathit{Dom}(\rho_2) = \{x \in \mathit{Vars} \mid \alpha \in \mathrm{TV}(\Gamma\, x)\} \Rightarrow\\
\quad\quad (\forall d \in [\![\vdash P_1]\!]_{\theta_1^*\,\sigma_1}\,\xi.\\
\quad\quad\quad\quad [\![\mathscr{A}_1 \vdash e :: \theta_1^*\,\sigma_1 \to \theta_1^*\,\sigma_2]\!](\rho \mathrel{+\!\!>} \rho_1) \bullet d \in [\![\vdash P_2]\!]_{\theta_1^*\,\sigma_2}\,\xi)\\
\quad\quad\quad\quad\quad\quad \Longleftrightarrow\\
\quad\quad (\forall d \in [\![\vdash P_1]\!]_{\theta_2^*\,\sigma_1}\,\xi.\\
\quad\quad\quad\quad [\![\mathscr{A}_2 \vdash e :: \theta_2^*\,\sigma_1 \to \theta_2^*\,\sigma_2]\!](\rho \mathrel{+\!\!>} \rho_1) \bullet d \in [\![\vdash P_2]\!]_{\theta_2^*\,\sigma_2}\,\xi)
\end{array}$$

Since the arrow $(\to)$ is a free predicate constructor the following equality is justified,

$$\forall \theta :: \mathit{Vars} \to \mathsf{Type}.\ \theta^*\,\sigma_1 \to \theta^*\,\sigma_2 = \theta^*\,(\sigma_1 \to \sigma_2)$$

from which the semantic definition of an arrow predicate yields the conclusion of the lemma.
Case $\sigma = \tau$, where τ is a ground type. Then the conclusion holds trivially.

We conclude by coinduction that the conclusion of the lemma holds for all typed assertions. □

Corollary 1
If a proposition $e :: \sigma ::: P$ is validated by extending a value assignment, ρ, at some ground type specialization $(\mathscr{A}, \tau) \in \mathrm{Gr}(\Gamma \vdash e :: \sigma)$ then it is validated for ρ extended at every such specialization.

Proof
The corollary is an immediate consequence of Lemma 1 and the enumerability of types. □

6 Soundness of P-logic

Soundness of a logic means that all of its inference rules are coherent with its semantics. An inference rule asserts a propositional implication of a consequent judgment from zero or more antecedent judgment forms.

6.1 Soundness of inference rules

An inference rule is *sound* if the implication it states is valid for a model of the logic. An implication is *valid* if it is true of a model under all type-compatible assignments to variables.

In this section, we offer a formal definition of soundness for inference rules, then proceed in section 6.2 to develop a reference frame model for the Haskell fragment, against which the soundness of rules can be checked. The reference frame model is itself specified by a Haskell module and is thus computable. Corollary 2 below will justify the choice of an arbitrary ground typing in this model, thereby reducing the problem of checking soundness of rules (8)–(18) to a finite, symbolic model checking problem, which is described in subsequent sections of the paper.

Definition 26
[Rule soundness]
Let Γ be a type environment which assigns a unique type variable to each term variable in its domain. A polymorphic rule of P-logic,

$$\frac{\Gamma; \Pi_1 \vdash_{\mathscr{P}} \Delta_1 \cdots \Gamma; \Pi_n \vdash_{\mathscr{P}} \Delta_n}{\Gamma; \Pi \vdash_{\mathscr{P}} \Delta}$$

is *sound* if there is a frame model, $\mathscr{D}$, such that under every predicate assignment, ξ

$$\mathscr{D}, \xi \models (\Pi_1 \Rightarrow \Delta_1) \Rightarrow \cdots \Rightarrow (\Pi_n \Rightarrow \Delta_n) \Rightarrow \Pi \Rightarrow \Delta$$

□

A rule may contain free term variables, which are implicitly universally quantified over the scope of the entire rule. In addition, the properties asserted in a rule are often represented by free predicate variables, also subject to implicit universal quantification over the rule.

Many rules of P-logic, in particular those characterizing the applicative structures and free term algebras of Haskell, are polymorphic, i.e. the types of terms and predicates in the rule contain at least one free type variable. Corollary 1 tells us that a polymorphic property can be observed at any type instance of a polymorphic type. In view of Definition 26, we also have the following as a corollary to Lemma 1.

Corollary 2
The soundness of a polymorphic rule of P-logic can be observed at any ground instance of its typing.

□

Not only does polymorphism allow the soundness of inference rules to be checked at an arbitrarily chosen type instance, but as a consequence of model-independence (see Lemma 8.2.5, (Mitchell, 2000)), soundness can be checked relative to any particular frame model.

6.2 A reference frame model

The model described here is an interpreter for the Haskell fragment whose semantics is given in Section 4. Although the semantic metalanguage used in defining the intepreter 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 (Wadler, 1992). The interpreter relies on the *Maybe* monad which was introduced in section 4.3.6 to model control flow among alternate match clauses.

Figure 6 contains a description of the underlying representation of value domains in the interpreter for the Haskell fragment considered in this paper. The interpretation function for expressions, *mE*, maps a typed expression and an environment to an untyped value in the domain V. The domain V is structured as a disjoint union of a distinguished element, *Bottom*, a set of tagged tuples (represented as finite lists) of values that model elements of data types, and a set of lists of value pairs that encode a trace representation of functions. The domain is partially ordered by a relation ($\sqsubseteq$), in which *Bottom* is a unique least element, strictly below every other element of V. The partial order extends pointwise to a partial ordering on tagged tuples. All of the interpreter functions are monotonic with respect to this order.

A list of pairs[9], *tc*, is the *trace* of a function if it satisfies the constraint

$$\forall (x_1, y_1), (x_2, y_2) \in tc.\, x_1 = x_2 \Rightarrow y_1 = y_2$$

The partial order ($\sqsubseteq$) extends to traces as follows:

$$FT(xs) \sqsubseteq FT(xs') \Leftrightarrow \forall x, y \in V.\, (x, y) \in xs \Rightarrow (y = Bottom \vee (x, y) \in xs')$$

A trace is *monotone* if $\forall x, x' \in V .x \sqsubseteq x' \Rightarrow f\, x \sqsubseteq f\, x'$. On finite domains, monotone functions preserve all limits and hence are continuous.

[9] Ordinarily, a trace would be defined as a set of ordered pairs. However, a list data structure, without repeated elements, is used in the interpreter to code a set.

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

— Domain of Values
data V          =     FT (V × V)          {- trace represenation of function values -}
                |     Tagged Name [V]     {- structured data -}
                |     Bottom              {- bottom element in a pointed domain -}

— Projection out of the Maybe monad
purify          ::  Maybe a → a
purify (Just x) =   x
purify Nothing  =   Bottom

— Alternation
([])            ::  (a → Maybe b) → (a → Maybe b) → (a → Maybe b)
(f [] g) x      =   case f x of
                     Nothing  →  g x
                     Just v   →  Just v
```

Fig. 6. Semantic operators used in the reference frame model.
Note that *purify* is analogous to the function *fromJust* defined in Haskell's standard prelude. However, when applied to the constructor *Nothing*, *purify* returns the symbolic value *Bottom*, a constructor in the data type *V*, whereas *fromJust* returns the semantic bottom of the data type.

The application operator ($\bullet$) in this frame model is

$$(\bullet) \ :: \ (V, V) \rightarrow V$$
$$FT(tc) \bullet v = purify\,(lookup\; v\; tc)$$

where $lookup :: Eq\ a \Rightarrow [(a, b)] \rightarrow Maybe\ b$ is defined in Haskell's standard Prelude and *purify* is defined in Fig. 6.

6.2.1 Frame sets for Haskell types

Figure 7 gives the underlying sets of type frames for the types modeled in the interpreter. The function mT calculates the frame set for a type. The second argument of mT is a "strictness value" used to indicate whether a frame set is pointed (noted by the argument value *Lazy*) or unpointed (noted by the argument value *Strict*).

The frame set for a data type is a set of representations of the saturated applications of its data constructors to elements of the frame sets of their argument types. These frame sets are either pointed or unpointed according to the strictness annotation, $s_{i,j}$ declared for each (j^{th}) argument of a data constructor C_i. Meanings of data constructors are given in Fig. 9.

The frame set of a finitary arrow type, $\tau_1 \rightarrow \tau_2$ is specified in terms of monotone *traces*, where $traces\ \tau_1\ \tau_2 \subset Powerset\,(|D_{\tau_1}| \times |D_{\tau_2}|)$ is the relation satisfying both the functionality and monotonicity constraints[10].

[10] The trace representation can also be extended to accommodate infinitary arrow types by adding the constraint that limits of directed sets are preserved, but as the Haskell fragment considered in this paper does not require infinitary types, the additional constraint has been omitted.

$$
\begin{array}{lcl}
mT & :: & T \to LS \to [V] \\
mT\ Triv\ Strict & = & [()] \\
\\
mT\ (T\ \tau_1 \cdots \tau_p)\ Strict & = & \\
\quad \bigcup_{i=1}^{n} \{ Tagged\ (name\ C_i)\ [t_{i,1}, \ldots, t_{i,k_i}] \mid t_{i,j} \leftarrow mT\ \sigma_{i,j}[\tau_1/\alpha_1, \ldots, \tau_p/\alpha_p]\ s_{i,j} & & \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{for } (1 \leqslant j \leqslant k_i) \} & & \\
\quad \text{where } Constr\ C_i\ [(s_{i,1}, \sigma_{i,1}) \cdots (s_{i,k_i}, \sigma_{i,k_i})] \in \Sigma_{T\ \alpha_1,\ldots,\alpha_p} \qquad \text{for } (1 \leqslant i \leqslant n) & & \\
\\
mT\ (\tau_1 \to \tau_2)\ Strict & = & \{ FT\ tc \mid tc \leftarrow traces\ \tau_1\ \tau_2 \} \\
\quad \text{where } \forall tc :: [(V,V)].\ tc \in traces\ \tau_1\ \tau_2 \Leftrightarrow & & \\
 & & (\forall t_1 \in (mT\ \tau_1\ Lazy).\ \exists t_2 \in (mT\ \tau_2\ Lazy).\ (t_1, t_2) \in tc) \wedge \\
 & & \forall (t_1, t_2), (t_1', t_2') \in tc.\ (t_1 \sqsubseteq t_1' \Rightarrow t_2 \sqsubseteq t_2') \wedge (t_1 = t_1' \Rightarrow t_2 = t_2') \\
\\
mT\ \tau\ Lazy & = & \{Bottom\} \cup (mT\ \tau\ Strict)
\end{array}
$$

Fig. 7. Frame model for a Haskell fragment: Type frame sets. (To compute type frame sets, a Haskell implementation represents sets by lists without repeated elements.)

6.2.2 Interpreting patterns

The semantics function *mP* interprets patterns, as computations in the *Maybe* monad. The data constructor *Nothing* in the codomain type designates failure of an attempt to match the pattern with an argument value; the data constructor *Just* injects a list of the component values extracted from an argument when it is deconstructed in a successful match.

Figure 6 also displays two combinators integral to modeling `case` expressions and patterns, called "fatbar" ($[\!]$) and *purify*. If m_1 and m_2 have type $(V \to Maybe\ V)$, then

$$
(m_1 \mathbin{[\!]} m_2)\ v = \begin{cases} (m_1\ v) & \text{if } (m_1\ v) = Just\ v' \\ (m_2\ v) & \text{otherwise.} \end{cases}
$$

This is precisely the sequencing behavior necessary for modeling `case` expressions. 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 8 and 9 display the semantics for patterns and expressions, *mP* and *mE*, respectively. These semantics specialize the abstract semantics of section 4 to the concrete representations given by the interpreter.

To confirm the assertion that the interpreter is a frame model, let's check the components specified in section 4.

- $\mathscr{D}$, a collection of typed frame objects, is comprised of the images of ground types under the mapping $\lambda\tau \text{->} mT\ \tau\ Lazy$, and subject to the partial order on the domain V, as defined in section 6.2.
- The application operation, $\bullet$, is defined in section 6.2.
- Given $f :: (\tau_1 \times \tau_2) \to \tau_3$,

$$
f^\sharp = FT\{(a, FT\ tc) \mid a \leftarrow mT(\tau_1),\ tc \leftarrow traces\ \tau_2\ \tau_3,\\
\forall b \in mT(\tau_2).\ mE\ f\ []\bullet (a,b) = FT\ tc \bullet b\}
$$

```
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 . snd) ps) vs)
                                     else Nothing
mP (Pcondata n ps) Bottom        =   Just Bottom
mP Pwildcard v                   =   Just []
mP (Ptilde p) v                  =   Just (case (mP p v) of
                                           Nothing   →
                                             take (length (fringe p)) (repeat Bottom)
                                           Just z      → z)

stuple                           ::  [V → Maybe[V]] → [V] → Maybe[V]
stuple [] []                     =   Just []
stuple (q : qs) (v : vs)         =   do v'   ←   q v
                                        vs'  ←   stuple qs vs
                                        return (v' ++ vs')
```

Fig. 8. Frame model for a Haskell Fragment: Patterns.

```
mE                        ::  E → Env → V
mE (Var x) ρ              =   ρ x
mE (Constr n ts) ρ        =   constrFun n ts []
mE (Case e ms) ρ          =   mcase ρ ms (mE e ρ)
mE (Abs (x :: τ) e ρ      =   FT [(v, mE e ρ[x ↦ v]) | v ← mT τ Lazy]
mE (App e₁ e₂ ρ)          =   let FT tc = mE e₁ ρ
                              in  purify (lookup (mE e₂ ρ) tc)
mE Undefined ρ            =   Bottom

constrFun n [] vs         =   Tagged n vs
constrFun n ((s,τ) : ts) vs = FT [(x,y) | x ← mT τ s,
                                          y ← constrFun n ts (vs ++ [x])]

match                     ::  Env → (P,E) → V → Maybe V
match ρ (p,e)             =   (mP p) ◇ (Just ∘ (\vs → mE e (extL ρ xs vs)))
                          where xs                       =  fringe p
                                extL ρ [] []             =  ρ
                                extL ρ (x : xs) (v : vs) =  extL (ρ[x↦v]) xs vs

mcase                     ::  Env → [(P,E)] → V → V
mcase ρ ms                =   purify ∘ (fatbarL (map (match ρ) ms))

fatbarL                   ::  [V → Maybe V] → V → Maybe V
fatbarL ms                =   foldr ([]) (\_ → Just Bottom) ms
```

Fig. 9. Semantics of a Haskell Fragment: Expressions.

– Given $g :: \tau_1 \to \tau_2 \to \tau_3$,

$$g^{\flat} = FT\{((a,b),c) \mid a \leftarrow mT(\tau_1),\, b \leftarrow mT(\tau_2),\, c = (mE\ g\ [\,]\ \bullet\ a)\ \bullet\ b\}$$

– For a data constructor, C_n, the interpreting semantic function is $c_n = mE\ C_n\ [\,]$.

- The pattern function for a data constructor, C_n is $c_n^M = mP \circ (Pcondata\ n)$.
- $c_n^{-1} = purify \bullet c_n^M$
- The interpreter uses the monadic operators defined for the Maybe monad in Haskell.
- The operator $\oplus$ is interpeted by the function *stuple*, which is defined in Fig. 8. Tuples of computations typed in the Maybe monad are represented as lists. The Kleisli composition ($\diamond$) and alternation operators are programmed analogously to their definitions in section 4.

6.3 Finite models for Haskell types

In this section, we consider the type constructions of the Haskell fragment, to show how each type or type construction can be represented by a finite type in which to model some rule of P-logic.

In checking any rule, the principle followed is to choose the simplest ground type possible to instantiate each type variable of a polymorphically typed term. Thus, for instance, to check rule (8) for abstraction introduction, notice that the rule is polymorphic in each of the two type meta-variables, τ_1 and τ_2, that are combined to form the arrow type. Thus we can choose to check the rule at the type $Triv \to Triv$, in which each type meta-variable has been instantiated to *Triv*, forming the simplest instance of an arrow type.

Notice that we do not require a recursive datatype constructor, such as *List*, to check soundness of the rules given in this paper. It is not necessary to choose a recursively defined type because none of the basic rules of P-logic concludes assertions that depend explicitly or implicitly on fixed-points. In particular, terms specific to data types occur only in rules (16)–(19). The terms in these rules contain no explicitly nested occurrences of data constructors and thus, soundness of these rules can be checked at a ground instance of the type

$$\textbf{data}\ StrictOption\ a = Cstrict\ !a \mid Clazy\ a$$

which includes both a data constructor sat-strict in its argument and a non-sat-strict constructor. We return in section 6.5.1 to take up the soundness of rule (19), in which patterns may implicitly be nested.

6.4 Modeling predicates

When a ground type instance of the terms in a rule has been chosen, the typing of every predicate in the rule is also determined. To check soundness of the rule, we simulate all type-compatible value assignments to term variables and predicate assignments to the predicate variables that occur in the rule.

At every type we have the predicates Univ, \$Univ, UnDef and \$UnDef. Notice however, that no information can be gotten from the assignment of Univ, as this predicate contains every element of the corresponding type's frame set, nor from the assignment of \$UnDef, which is unsatisfied by any element of the frame set.

In addition to the interpretations of $Univ and UnDef, interpretations are required for predicates at a particular type. For instance, for the arrow type, *Triv* → *Triv*, the needed predicate interpretations are:

$$
\begin{aligned}
\$(\$\mathsf{Univ} \to \$\mathsf{Univ}) &= \{FT\ [((\,), (\,))]\}\\
\$(\$\mathsf{Univ} \to \mathsf{Univ}) &= \{FT\ [((\,), Bottom)],\ FT\ [((\,), (\,))]\}\\
\$(\mathsf{Univ} \to \$\mathsf{Univ}) &= \{FT\ [(Bottom), (\,)),\ ((\,), (\,))]\}\\
\$(\mathsf{Univ} \to \mathsf{Univ}) &= \{FT\ [(Bottom, Bottom), ((\,), Bottom)],\\
&\qquad FT\ [(Bottom, Bottom), ((\,), (\,))],\ FT\ [(Bottom, (\,)), ((\,), (\,))]\}
\end{aligned}
$$

Notice that the non-monotonic function trace *FT* [(*Bottom*, ( )), (( ), *Bottom*)] is not generated as a member of any predicate interpretation.

The interpretation of $Univ at the type *Triv* → *Triv* is the union of the strong, arrow-specific interpretations listed above. The interpretation of any weak predicate is just the union of its strong interpretation with the singleton set, {*Bottom*}.

6.5 Automated model checking of inference rules

The interpreter given in section 6.2 provides a machine-executable frame model for the Haskell fragment. Using the types described in section 6.3, it is straightforward to calculate the elements of each type frame set. In this section, we describe how this executable model has been used to check the soundness of polymorphic inference rules by calculation.

An initial step in model-checking a polymorphic rule is the choice of a type instance, justified by Corollary 2. Instantiating each type variable at the type *Triv* meets this requirement. This is sufficient for rules (8)–(11). For rules (16)–(20), we choose the data type *StrictOption Triv*.

A valuation assignment for the free term variables occurring in a rule simply binds each variable to an element of the frame set corresponding to the type of the variable. Universal quantification over valuation assignments is realized by iterating through all possible value assignments, for each variable independently, at the finite type in which the rule is to be checked.

Similarly, a predicate assignment binds a subset of the type frame set to each predicate variable that occurs free in a rule. Quantification over predicate assignments is realized by iterating over all type-compatible predicate assignments.

At each valuation and each predicate assignment to the free variables occurring in a rule, the truth of the propositional implication realized by that particular instance of the rule is checked. A proposed rule is sound if all such checks succeed at the selected type; unsound if any such rule instance is false.

For example, recall the polymorphic rule (9):

$$\frac{\Pi \vdash_{\mathscr{P}} e' ::: P \qquad e\, e' ::: Q \vdash_{\mathscr{P}} \Delta}{\Pi, e ::: \$(P \to Q) \vdash_{\mathscr{P}} \Delta}$$

It can be checked under the typing assignment $e' :: Triv$, $e :: Triv \to Triv$, $P, Q :: Pred\ Triv$. For each particular valuation assignment and predicate assignment, we calculate the weakest context assumption, Π, and the strongest entailment, Δ, for which both of the rule's antecedent clauses are true. Then, using these assignments and the calculated context assumption and entailment propositions, the truth of the rule's consequent is checked, using the interpretations provided by the reference frame model to evaluate Haskell terms. The process described here is fully automated by a Haskell program.

In checking rule (9), the assumption calculated to validate the antecedents provides a binding for a term (the meta-variable, e') to which e is applied in an assumption of the second antecedent. Even though this application is not explicit in the hypothesis of the consequent, the assumed binding is present in the valuation of Π, and provides support for the calculated entailment. This succeeds under each valuation and predicate assignment for which the antecedents of the rule could be validated; thus the rule is deemed sound.

However, when the hypothesis in the consequent of the rule is weakened, as in

$$(\mathit{Unsound}) \qquad \frac{\Pi \vdash_{\mathscr{P}} e' ::: P \qquad e\,e' ::: Q \vdash_{\mathscr{P}} \Delta}{\Pi,\, e ::: (P \rightarrow Q) \vdash_{\mathscr{P}} \Delta}$$

the rule is found to be false under the valuation assignment $[(e', \mathit{Triv}), (e, \mathit{Bottom})]$ and the predicate assignment $[P = \mathsf{Univ},\ Q = \$\mathit{Triv}]$. Under these assignments, we calculate from the antecedents a weakest context constraint $(e', ()) \in \Pi$ and a strongest entailment constraint $(e\,e', ()) \in \Delta$. These constraints are not both satisfiable in the consequent, under the semantics of application. Thus, had the modified rule been proposed as a rule of P-logic, it would have been found unsound by automated model checking and rejected.

6.5.1 Soundness of rule (19)

Rule scheme (19), which is repeated below, is polymorphic in the types of the variables in the pattern of a case branch.

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

It is not polymorphic in the type of a pattern itself, however, and thus soundness of the rule cannot be checked at an arbitrarily chosen, "small" type. Since the rule scheme accommodates nested patterns, we shall prove it sound by inducting on the structure of a pattern. At base cases for the induction, and also for the induction steps, the proof will make use of model-checking to verify that these cases are valid under all well-typed value assignments to pattern variables and to predicate variables. Model-checking can be done at a set of "small" type instances that are assumed for the variables occurring in the pattern's fringe.

Recall that the predicate associated with a pattern is calculated by:

$$\pi(p)\ \mathit{preds} \quad = \quad \mathit{fst}\ (\mathit{patPred}\ p\ \mathit{preds})$$

The following lemma relates the sequence of predicate arguments consumed by the application *patPred p preds* to the sequence of variables bound in a pattern, *fringe p*.

Lemma 2

[Associating predicates with the fringe of a pattern]

Let p be a pattern and $preds = [P_1, P_2, \ldots]$ be a sequence of predicate formulas such that $length\ preds \geqslant length\ (fringe\ p)$. Then

$$\begin{array}{l} patPred\ p\ preds = \\ \quad (fst\ (patPred\ p\ (take\ (length\ (fringe\ p))\ preds)),\ drop\ (length\ (fringe\ p))\ preds) \end{array}$$

The proof of Lemma 2 is by induction on the structure of a pattern. Details of the proof are given in the Appendix.

Definition 27

[Implication ordering of predicates]

Let $(\preceq)\ \subseteq\ Pred \times Pred$ be the smallest relation transitively closed under the following:

$$\begin{array}{l} P \preceq \mathsf{Univ} \\ \$\mathsf{UnDef} \preceq P \\ \$P \preceq P \\ P \preceq Q \Rightarrow \$P \preceq \$Q \\ P_1 \preceq Q_1 \Rightarrow \cdots \Rightarrow P_k \preceq Q_k \Rightarrow C^{(k)}\, P_1 \cdots P_k \preceq C^{(k)}\, Q_1 \cdots Q_k \end{array}$$

A ramification of the implication ordering is that in every ground type assignment, $\mathscr{A}$, and for all $\mathscr{A}$-compatible assumptions, Π, if $t :: \tau$ is a well-typed term and P and Q are $(\preceq)$-related predicates of type $Pred\ \tau$, then

$$P \preceq Q \Rightarrow \Pi \vdash_{\mathscr{P}} t ::: P \Rightarrow \Pi \vdash_{\mathscr{P}} t ::: Q$$

Definition 28

[Substitution of predicates for pattern variables]

$$\begin{array}{lll} subst :: Pattern \rightarrow [(Var, Pred)] \rightarrow Pred & & \\ x\ \text{`}subst\text{`}\ [(x, P)] & = P & \\ \_\ \text{`}subst\text{`}\ prs & = \mathsf{Univ} & \\ {\sim}p\ \text{`}subst\text{`}\ [(x_1, \mathsf{Univ}), \ldots, (x_k, \mathsf{Univ})] = \mathsf{Univ} & & \textbf{where}\ [x_1, \ldots, x_k] = fringe\ p \\ {\sim}p\ \text{`}subst\text{`}\ prs & = p\ \text{`}subst\text{`}\ prs & \\ C_n\ \text{`}subst\text{`}\ prs & = C_n & \\ C_n\, p_1 \cdots p_k\ \text{`}subst\text{`}\ prs & = & \\ \quad \texttt{let}\ \ P_1 = p_1\ \text{`}subst\text{`}\ (take\ (length\ (fringe\ p_1))\ prs) & & \\ \quad \phantom{\texttt{let}}\ \ C_n\ P_2 \cdots P_k = C_n\, p_2 \cdots p_k\ \text{`}subst\text{`}\ (drop\ (length\ (fringe\ p_1))\ prs) & & \\ \quad \texttt{in}\ C_n\ P_1\ P_2 \cdots P_k & & \end{array}$$

Lemma 3

[Binding of predicates for pattern variables]

Let p be a pattern and $preds = [P_1, P_2, \ldots]$ be a sequence of predicate formulas such that $length\ preds \geqslant length\ (fringe\ p)$. Since $fringe\ p$ can contain no repeated occurrences of variables, the association list, $zip\ (fringe\ p)\ (take\ (length\ (fringe\ p))\ preds)$, can be interpreted as a substitution of predicates for variables. The following predicate relation holds for all predicate-derived patterns:

$$\pi(p)\ preds \preceq p\ \text{‘}subst\text{‘}\ zip\ (fringe\ p)\ (take\ (length\ (fringe\ p))\ preds) \qquad (29)$$

Proof

By induction on the structure of a pattern. Details of the proof are given in the Appendix. □

When the terms of a sequent have types restricted to *Triv* and the arrow types that can be formed with *Triv* as a base type, all frame models that distinguish the bottom element from the non-bottom element of *Triv* are equivalent. When data types are allowed, however, the choice of a frame set having a finite cardinality at its base types may affect the validity or satisfiablity of a sequent with respect to that model. Note, however, that recursive data types are not required in the languge fragment we have considered, so a data type has only finite cardinality. And as the consequent of rule (19) doesn't specify the arity of constructors that may occur in a pattern, we might imagine that a data type of bounded size (number and arity of constructors) could suffice to establish its validity or satisfiablility. That is, should there be a counterexample to the validity (or satisfiability) of this sequent, there must be such in a data type of bounded size.

In fact, we can choose as a prototypical data type *StrictOption Triv*. This type has enough constructors to discriminate matching and non-matching patterns in a case expression and it includes constructors both strict and non-strict in an argument position.

Lemma 4

The following rule scheme, which is a modification of rule scheme (19), is sound.

$$\frac{\Pi,\ x_1\ :::\ P_1, \cdots, x_k\ :::\ P_k \vdash_{\mathscr{P}} t\ :::\ Q}{\Pi \vdash_{\mathscr{P}} \{p \texttt{ -> } t\}\ :::\ p\ \text{‘}subst\text{‘}\ zip\ (fringe\ p)\ [P_1 \cdots P_k] \rightarrow \texttt{\$Just}\ Q}$$

where $[x_1, \ldots, x_k] = fringe\ p$.

Proof

This rule is model-checked at the type *StrictOption Triv* → Maybe *Triv*. Sat-strictness or non-sat-strictness of the data constructors has no effect on the substituted predicate pattern. □

Theorem 1

Rule scheme (19) is sound.

Proof

The conclusion follows directly from Lemma 2 and Lemma 4. As a consequence of the predicate ordering $\pi(p)\ P_1 \cdots P_k \preceq p\ \text{‘}subst\text{‘}\ zip\ (fringe\ p)\ [P_1 \cdots P_k]$, under a predicate interpretation and value assignment for which the consequent of (19) is valid, the consequent of the modified rule of lemma 4 is also valid. Thus soundness of the modified rule implies soundness of rule (19). □

7 Related work

As part of the *Programatica* project at the Pacific Software Research Center, we are developing both a formal basis for reasoning about Haskell programs, and automated tools for mechanizing such reasoning.

Simon Thompson's early effort to give a verification logic (Thompson, 1995) 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 data type. The meanings of such quantifiers are extra-logical; they cannot be defined in the logic itself.

Sparkle (de Mol *et al.*, 2001) is a verification tool for *Clean* (Plasmeijer & van Eekelen, 1999), 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 characterizing properties of unbounded terms of a specific abstract syntax. From the *Stratego* language[11] we learned of data constructor congruences, whereby the initial-algebra property of a freely constructed data type 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 in a similar way to synthesize predicates satisfied by constructed terms out of 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 (Owe, 1993), 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 (Larsen, 1990) 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 (Huth *et al.*, 2001) 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

[11] For more information, please refer to the Stratego homepage: `www.stratego-language.org`.

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 (Owe, 1993; Gumb & Lambert, 1996; Gumb & Lambert, 1997; Cheng & Jones, 1991; Farmer, 1995; Gries & Schneider, 1995; Konikowska *et al.*, 1991). For an excellent overview, the interested reader should consult Farmer (Farmer, 1995).

8 Conclusions

The language fragment which concerns us here is the part of Haskell 98 that has to do with demand: pattern-matching. We have presented a two succinct formalisms that specify the denotational and axiomatic semantics of Haskell pattern-matching, which is a surprisingly complex aspect of the language. Pattern-matching in ML (Milner *et al.*, 1997), for example, is comparatively much simpler. The relative complexity of Haskell's pattern-matching arises chiefly from Haskell's default lazy evaluation and from the possibility that irrefutable patterns may be embedded as sub-patterns. Pattern-matching is essentially an eager activity, and is thus harmonious with ML's eager semantics.

The first part of this paper reports on part of a semantics for the whole of Haskell 98, some of which has been reported elsewhere (Harrison *et al.*, 2002). One hurdle to overcome when attempting to write a formal semantics for a large language is identifying an appropriate semantic framework in which to specify the entire language. Haskell 98 has a number of features which have been specified at varying levels of formality operationally, denotationally, or informally: type classes and overloading, polymorphism, polymorphic recursion, and mixed evaluation to name just a few. The problem we immediately confronted was: what is a sufficiently expressive framework in which to specify the whole language? Because we wished to use this semantics to evaluate the faithfulness of *P*-logic, we narrowed our selection to denotational semantics.

However, we still faced many choices. Should we take, for instance, a purely domain-theoretic approach? It was felt that such an approach, while clearly sufficient in terms of expressiveness, would lack the desired level of abstraction for a standard semantics. In other words, domain-theoretic models include considerably more concrete representation detail than we desired. Indeed, there are many suitable varieties of domains to model Haskell types, and calling any one of these "standard" could hardly avoid being seen as an arbitrary choice.

Ultimately, we fastened onto frame semantics as a suitably abstract foundation for Haskell 98. The underlying representations of frame objects (i.e., what would be individual cpos in a domain-theoretic model) are left unspecified, constrained only by the extra structure and their axiomatizations. This representation independence was extremely useful in the proof of soundness, allowing us to use model-checking over finite models of types for many rules.

Another virtue of frames as a semantic basis for Haskell 98 is their close connection to the semantics of ML polymorphism. Ohori (Ohori, 1989b; Ohori, 1989a) demonstrated that frame semantics for simply-typed lambda calculae may be conservatively extended in a compelling, elegant, and natural way to a semantics for (first-order) polymorphism – precisely the variety of polymorphism found in functional programming languages like Haskell or ML. Ohori's semantics has a further virtue as a basis for Haskell: the type information within denotations allows other Haskell features to be captured. Overloading and polymorphic recursion—both Haskell features in need of illumination—can be neatly expressed in Ohori's setting, although we leave this part of Haskell 98's semantics to a sequel.

P-logic is a verification logic for all of Haskell 98, although we have only shown here the part essential to expressing Haskell's fine control of demand. With its two modalities, one can formulate properties in *P*-logic more precisely than would be possible if predicates could be written in only a single modality. Restricting predicates to the weak modality would result in a partial-correctness logic, as every predicate would be satisfied by bottom-denoting expressions as well as those denoting normal values. If all predicates were restricted to the strong modality, only properties of provably terminating computations could be verified. In *P*-logic, one can express that a function is total; yet not every property entails the obligation to prove that a denotation is non-bottom.

The proof rules of *P*-logic are sufficiently subtle that their soundness cannot easily be confirmed by a quick, visual inspection. However, we were able to mechanize the most detailed parts of a soundness proof by employing an executable frame model for Haskell's semantics to systematically check polymorphic proof rules at a simple type. The meta-theory that supports this automatic soundness checking is one of the contributions of this paper.

Acknowledgments

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.

Appendix

This appendix contains proofs of Lemmas 2 and 3. Pattern terms will be presented in their concrete syntax except when they appear as arguments of the function *patPred* in the proof of Lemma 2, where abstract syntax is used.

Lemma 2

Let p be a pattern and $preds = [P_1, P_2, \ldots]$ be a finite sequence of predicate formulas such that $length\ preds \geqslant length\ (fringe\ p)$. Then

$$patPred\ p\ preds =$$
$$(fst\ (patPred\ p\ (take\ (length\ (fringe\ p))\ preds)),\ drop\ (length\ (fringe\ p))\ preds)$$

Proof
By induction on the structure of a pattern. Each equation in the definition of *patPred* corresponds to one such case.

Case $p = x$, an individual pattern variable.

$$\begin{aligned}&patPred\ (Pvar\ x)\ (P\ :preds)\\ &= (P, preds)\\ &= (fst\ (patPred\ (Pvar\ x)\ [P]),\ drop\ (length\ [x])\ (P\ :preds))\end{aligned}$$

Case $p = \_$, a wildcard pattern.

$$\begin{aligned}&patPred\ (Pwildcard)\ preds\\ &= (\mathsf{Univ}, preds)\\ &= (fst\ (patPred\ (Pwildcard)\ []),\ drop\ (length\ [])\ preds)\end{aligned}$$

Case $p = {\sim}p'$, an irrefutable pattern. Recall that $fringe\ {\sim}p' = fringe\ p'$. There are two subcases: If $take\ (length\ (fringe\ p'))\ preds = [\mathsf{Univ}, \ldots, \mathsf{Univ}]$ then

$$\begin{aligned}&patPred\ (Ptilde\ p')\ preds\\ &= (\mathsf{Univ}, drop\ (length\ (fringe\ p'))\ preds)\\ &= (fst\ (patPred\ (Ptilde\ p')\ (take\ (length\ (fringe\ p'))\ preds)),\\ &\quad\ drop\ (length\ (fringe\ p'))\ preds)\end{aligned}$$

Otherwise,

$$patPred\ (Ptilde\ p')\ preds = patPred\ p'\ preds$$

for which we assume the assertion holds as a hypothesis of induction.

Case $p = C_n\, p_1 \ldots p_k$, a pattern formed of a constructor applied to k arguments. (The enumeration index, n, is assumed to be unique to the constructor symbol.) Assume as a hypothesis of induction that the lemma holds for the first sub-pattern, p_1. To prove the assertion of the lemma for constructor patterns, we appeal to an inner-level induction on the number of arguments, k. Note that $fringe\ (C_n\, p_1 \ldots p_k) = foldr\ (+\!\!+)\ []\ [fringe\ p_1, \ldots, fringe\ p_k]$.
Subcase $k = 0$. Then

$$\begin{aligned}&patPred\ (Pcondata\ n\ [])\ preds\\ &= (Strong\ (ConPred\ n\ []),\ preds)\\ &= (fst\ (patPred\ (Pcondata\ n\ [])\ []),\ preds)\\ &= (fst\ (patPred\ (Pcondata\ n\ (take\ 0\ preds))),\ drop\ 0\ preds)\end{aligned}$$

Note that this satisfies the lemma.
Subcase $k = j + 1$. Then

$$\begin{aligned}&patPred\ (Pcondata\ n\ ((s_1, p_1)\ :\ pats))\ preds\\ &= \mathtt{let}\ (pr_1, preds_1)\ \ =\ \ patPred\ p_1\ preds\\ &\qquad\quad (prs, preds')\ \ =\ \ patPred\ (Pcondata\ n\ pats)\ preds_1\\ &\quad\ \mathtt{in}\ \ (Strong\ (ConPred\ n\ (ifStrict\ s_1\ pr_1\ :\ extract\_pr\_list\ prs)),\ preds')\end{aligned}$$

From the definitions in Figure 5 we note that either $ifStrict\ s_1\ p_1 = p_1$ or, in case $s_1 = Strict$ and p_1 is not already a strong predicate, $ifStrict\ s_1\ p_1$ lifts the predicate p_1

to the strong modality. In either case, any term that satisfies *ifStrict* s_1 p_1 is assured to satisfy p_1.

As a hypothesis of the inner-level induction, assume that the lemma holds for the pattern *Pcondata n pats*, where *length pats* = *j*. That is,

$$\begin{array}{l} patPred\ (Pcondata\ n\ pats)\ preds_1 = \\ \quad (fst\ (patPred\ (Pcondata\ n\ pats) \\ \qquad\qquad (take\ (length\ (fringe\ (Pcondata\ n\ pats)))\ preds_1)), \\ \quad drop\ (length\ (fringe\ (Pcondata\ n\ pats)))\ preds_1) \end{array}$$

Assume as a hypothesis of the top-level induction that the lemma holds for the first pattern, p_1, giving

$$\begin{array}{rl} patPred\ p_1\ preds & = (fst\ (patPred\ p_1\ (take\ (length\ (fringe\ p_1))\ preds), \\ & \qquad drop\ (length\ (fringe\ p_1))\ preds) \\ & = (pr_1, preds_1) \end{array}$$

Observing that

$$\begin{array}{l} length\ (fringe\ (Pcondata\ n\ ((s_1, p_1) : pats))) = \\ \qquad length\ (fringe\ p_1) + length\ (fringe\ (Pcondata\ n\ pats)) \end{array}$$

and using the definition of *patPred* from Fig. 5, a straightforward algebraic manipulation shows that the lemma is satisfied for a constructor pattern that has *k* arguments. This completes the inner-level induction.

Having discharged the proof for all cases, it follows by induction on the structure of patterns that the assertion of the lemma holds for all patterns. □

Lemma 3

Let *p* be a pattern and *preds* = $[P_1, P_2, \ldots]$ be a finite sequence of predicate formulas such that *length preds* $\geqslant$ *length* (*fringe p*). Since *fringe p* can contain no repeated occurrences of variables, the association list, *zip* (*fringe p*) (*take* (*length* (*fringe p*)) *preds*), can be interpreted as a substitution of predicates for variables. The following predicate relation holds for all predicate-derived patterns:

$$\pi(p)\ preds \preceq p\ \text{`}subst\text{'}\ zip\ (fringe\ p)\ (take\ (length\ (fringe\ p))\ preds)$$

Proof

By induction on the structure of a pattern.

Case $p = x$, a variable. Then *fringe* $p = [x]$, $\pi(p)\ [P_1, \ldots] = P_1$ and p *'subst'* $[(x, P_1)] = P_1$. Since $P_1 \preceq P_1$, Lemma 3 is satisfied.

Case $p = \_$, the wildcard pattern. Then $\pi(p)\ preds = \mathsf{Univ} \preceq \_$ *'subst'* $[]$, and the equation in Lemma 3 is satisfied.

Case $p = {\sim}p'$, an irrefutable pattern. Then *fringe* p = *fringe* p'. There are two cases. If *take* (*length* (*fringe p*)) *preds*) = $[\mathsf{Univ}, \ldots, \mathsf{Univ}]$ then $\pi(p)\ preds = \mathsf{Univ}$ and the equation in Lemma 3 is satisfied (trivially). Otherwise, $\pi(p)\ preds = \pi(p')\ preds$. As a hypothesis of induction, we assume that Lemma 3 holds for for the sub-pattern, p'. Therefore Lemma 3 holds also for the irrefutable pattern.

Case $p = C^{(k)}\, p_1 \cdots p_k$, a constructor pattern. Then

$$\mathit{fringe}\ p = \mathit{foldr}\ (++)\ []\ [\mathit{fringe}\ p_1, \ldots, \mathit{fringe}\ p_k]$$

To carry out the proof for a constructor pattern, we introduce an inner level of induction over the number of arguments, k.

Subcase $k = 0$. $\pi(C_n)\ [] = \$C_n \preceq C_n = C_n$ *'subst'* $[]$

Subcase $k = j + 1$ As a hypothesis of the outer level, structural induction, assume the assertion of the Lemma 3 holds for the first argument pattern,

$$\begin{aligned}&\pi(p_1)\ (\mathit{take}\ (\mathit{length}\ (\mathit{fringe}\ p_1))\ \mathit{preds}) \preceq \\ &\quad p_1\ \text{`}\mathit{subst}\text{'}\ \mathit{zip}\ (\mathit{fringe}\ p_1)\ (\mathit{take}\ (\mathit{length}\ (\mathit{fringe}\ p_1))\ \mathit{preds})\end{aligned}$$

As a hypothesis of the induction on the number of arguments, assume that Lemma 3 holds for the j-argument constructor pattern:

$$\begin{aligned}&\pi(C_n\, p_2 \cdots p_k)\ (\mathit{drop}\ (\mathit{length}\ (\mathit{fringe}\ p_1))\ \mathit{preds}) \preceq \\ &\quad C_n\, p_2 \cdots p_k\ \text{`}\mathit{subst}\text{'}\ \mathit{zip}\ (\mathit{fringe}\ (C_n\, p_2 \cdots p_k)) \\ &\qquad\qquad\qquad\qquad (\mathit{drop}\ (\mathit{length}\ (\mathit{fringe}\ p_1))\ \mathit{preds})\end{aligned}$$

A necessary condition for the above partial order is that the predicate relation holds for the pattern predicated derived from the argument patterns:

$$\begin{aligned}&\pi(p_i)\ (\mathit{take}\ (\mathit{length}\ (\mathit{fringe}\ p_i)) \\ &\qquad (\mathit{drop}\ (\mathit{sum}\ [\mathit{length}\ (\mathit{fringe}\ p_1), \ldots \mathit{length}\ (\mathit{fringe}\ p_{i-1})])\ \mathit{preds})) \\ &\preceq \\ &p_i\ \text{`}\mathit{subst}\text{'}\ \mathit{zip}\ (\mathit{fringe}\ p_i)\ (\mathit{take}\ (\mathit{length}\ (\mathit{fringe}\ p_i)) \\ &\qquad (\mathit{drop}\ (\mathit{sum}\ [\mathit{length}\ (\mathit{fringe}\ p_1), \ldots \mathit{length}\ (\mathit{fringe}\ p_{i-1})])\ \mathit{preds})) \\ &\text{for all } i \in [2..k]\end{aligned}$$

These conditions, together with the assumed ordering relation for the predicate derived from pattern p_1 is sufficient to establish Lemma 3 for the k-argument constructor pattern.

Thus the conclusion of the lemma follows by structural induction. □

References

Barr, M. and Wells, C. (1990) *Category Theory for Computing Science*. Prentice Hall.

Cheng, J. H. and Jones, C. B. (1991) On the usability of logics which handle partial functions. *Proceedings of the Third Refinement Workshop*. Springer-Verlag.

de Mol, M., van Eekelen, M. and Plasmeijer, R. (2001) Theorem proving for functional programmers. *Proceedings of the 13th International Workshop on the Implementation of Functional Programming Languages (IFL'01).*

Farmer, W. M. (1995) Reasoning about partial functions. *Erkenntnis*, **43**: 279–294.

Faxen, K.-F. (2002) A static semantics for haskell. *J. Funct. Program.* **12**(4&5): 295–357.

Girard, J.-Y. (1972) *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. Thèse d'état, University of Paris VII.

Girard, J.-Y. (1989) *Proofs and Types*. Cambridge Tracts in Theoretical Computer Science, vol. 7. Cambridge University Press.

Gries, D. and Schneider, F. B. (1995) Avoiding the undefined by underspecification. van Leeuwen, J. (ed.), *Computer Science Today: Recent Trends and Developments: Lecture Notes in Computer Science 1000*, pp. 366–373. Springer-Verlag.

Gumb, R. D. and Lambert, K. (1996) A free logical foundation for nonstrict functions. *Proceedings of the CADE-13 Workshop on the Mechanization of Partial Functions*, pp. 39–46.

Gumb, R. D. and Lambert, K. (1997) Definitions in nonstrict positive free logic. *Modern Logic*, **7**: 25–55.

Gunter, C. A. (1992) *Semantics of Programming Languages: Structures and Techniques*. MIT Press.

Harper, R. and Mitchell, J. C. (1993) On the type structure of standard ml. *ACM Trans. Program. Lang. & Syst. (TOPLAS)*, **15**(2): 211–252.

Harrison, W., Sheard, T. and Hook, J. (2002) Fine control of demand in Haskell. *6th International Conference on the Mathematics of Program Construction (MPC 2002): Lecture Notes in Computer Science 2386*, pp. 68–93. Springer-Verlag.

Hindley, R. J. (1969) The principal type scheme of an object in combinatory logic. *Trans. Am. Math. Soc.* **146**: 29–60.

Hudak, P. (2000) *The Haskell School of Expression: Learning Functional Programming through Multimedia*. Cambridge University Press.

Hudak, P., Peterson, J. and Fasel, J. (2000) *A Gentle Introduction to Haskell*. Haskell language tutorial online at `www.haskell.org/tutorial`.

Huth, M., Jagadeesan, R. and Schmidt, D. (2001) Modal transition systems: A foundation for three-valued program analysis. *Proceedings of the European Symposium on Programming (ESOP 2001): Lecture Notes in Computer Science 2028*. Springer-Verlag.

Jones, M. P. (1999) Typing haskell in haskell. *Proceedings of the 1999 Haskell Workshop* pp. 68–78. Published in Technical Report UU-CS-1999-28, Department of Computer Science, University of Utrecht.

Konikowska, B., Tarlecki, A. and Blikle, A. (1991) A three-valued logic for software specification and validation. *Fundamenta Informaticae*, **XIV**: 411–453.

Larsen, K. G. (1990) Modal specifications. *Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems: Lecture Notes in Computer Science 407*, pp. 232–246. Springer-Verlag.

MacQueen, D. B., Plotkin, G. and Sethi, R. (1984) An ideal model for recursive polymorphic types. *Information and Control*, **71**(1/2).

Milner, R. (1978) A theory of type polymorphism in programming languages. *J. Comput. & Syst. Sci.* **17**(3): 348–375.

Milner, R., Tofte, M., Harper, R. and MacQueen, D. (1997) *The Definition of Standard ML (Revised)*. The MIT Press.

Mitchell, J. C. and Harper, R. (1988) The essence of ML. *Conference Record of the 15th Annual ACM Symposium on Principles of Programming Languages (POPL 1988)*, pp. 28–46.

Mitchell, J. C. (2000) *Foundations for Programming Languages*. Third edn. MIT Press.

Ohori, A. (1989a) A Simple Semantics for ML Polymorphism. *Proceedings of the 4th International Conference on Functional Programming Languages and Computer Architecture (FPCA)*, pp. 281–292.

Ohori, A. (1989b) *A Study of Semantics, Types, and Languages for Databases and Object-oriented Programming*. PhD thesis, University of Pennsylvania.

Owe, O. (1993) Partial logics reconsidered: A conservative approach. *Formal Aspects of Computing*, **5**(3): 208–223.

Peyton Jones, S. (ed). (2003) *Haskell 98 Language and Libraries : The Revised Report*. Cambridge University Press.

Plasmeijer, R. and van Eekelen, M. (1999) Functional programming: Keep it CLEAN: A unique approach to functional programming. *ACM SIGPLAN Notices*, **34**(6): 23–31.

Reynolds, J. C. (1974) Towards a theory of type structure. *Programming Symposium: Lecture Notes in Computer Science 19*, pp. 408–425. Springer-Verlag.

Schmidt, D. A. (1986) *Denotational Semantics: A Methodology for Language Development*. Allyn and Bacon.

Smyth, M. B. and Plotkin, G. D. (1982) The category-theoretic solution of recursive domain equations. *SIAM J. Comput*. **11**(4): 761–783.

Thompson, S. (1995) A logic for miranda, revisited. *Formal Aspects of Computing*, **7**: 412–429.

Thompson, S. (1999) *Haskell: The Craft of Functional Programming*. Addison-Wesley.

Wadler, P. (1992) The essence of functional programming. *Conference Record of the 19th Annual ACM Symposium on Principles of Programming Languages (POPL 1992)*, pp. 1–14.