P-logic: property verification for Haskell programs

Richard B. Kieburtz

OGI School of Science & Engineering
Oregon Health & Science University
Beaverton, Oregon, USA

Version of February 20, 2002

Abstract

Proof-supported logical verification of program properties has been a topic of research interest for more than 30 years. The feasibility of proof construction as a verification technique has been demonstrated through many examples of its application, yet it remains a technique rarely used in practice for a variety of reasons, both technical and sociological. The lack of verification logics for modern programming languages remains a strong deterrent to the use of proof-supported verification.

This paper introduces *P*-logic, a verification logic for Haskell. *P*-logic is a modal mu-calculus that supports direct expression of recursively-defined properties of complex data structures. The term language of *P*-logic is Haskell. Logical assertions expressed in *P*-logic can be interleaved among definitions in a Haskell program text and can incorporate term variables bound in the program context. Properties of finite and infinite data structures can be expressed.

The paper describes syntax, proof rules, and semantics of *P*-logic and provides a few examples of its use.

1 Introduction

The *Programatica* project (*http://www.cse.ogi.edu/PacSoft/projects/programatica/default.htm*) is exploring a variety of means for validation of Haskell programs, including a comprehensive verification logic for Haskell. A particular requirement of a verification logic is that it must provide for direct expression of properties of programs. The term language of the logic should be the programming language itself, rather than some simplified modeling language. The logic should provide syntax with which to express properties of structured data and codata naturally and directly. When a programming language supports recursive definitions, its verification logic should have rules for reasoning about properties consequent to these definitions. Particularly for Haskell, whose syntax and type system do not discriminate between values and latent computations, the verification logic must support such discriminations. These desired properties of a verification logic are not found in generic, first-order or higher-order predicate logics.

This paper describes *P*-logic, a verification logic for Haskell. *P*-logic is a modal mu-calculus that supports direct expression of recursively-defined properties of complex data structures. In a modal logic, the intended domain of interpretations is not simply sets, but a family of sets with a particular structure. The modalities of *P*-logic dictate interpretations over sets with the structure of specific Haskell datatypes. This logic allows more concise, precise and exact assertions of many kinds of properties than has been possible with other logics with which we are familiar. *P*-logic also accommodates Haskell's lazy evaluation semantics with predicates that specifically either do or do not require a Haskell expression to be well-defined for satisfaction of the predicate.

P-logic is intended to be used in reasoning about programs that have independently been determined to be well-typed and to satisfy the constraints of static-semantics correctness. That is, it is intended for programs that compile correctly. This allows soundness criteria for the logic to be stated relative to a standard model for Haskell; a universe of pointed c.p.o.s that model the semantics of Haskell's types.

In this paper, many rules of *P*-logic have been omitted. Many of the omitted rules capture constructs of Haskell that allow programs to be written more concisely, but are otherwise inessential. These include strict data constructors, deferred-match patterns, syntax for guarded patterns, and mutually recursive let and where definitions. No rules are given here for the algebras of predefined primitive types, nor for the predefined operators of the IO monad.

The structure of the paper is as follows. Section 2 gives an informal introduction to the structure of *P*-logic and the modalities of Haskell terms that are embedded in the logic. Section 3 defines both the abstract and concrete syntax of the logic, gives typing rules, and provides a few examples. In Section 4, many of the inference rules are given. Section 5 gives rules applicable to recursive definitions in Haskell and for recursively-defined properties. The semantics of the logic and the notion of soundness are discussed in Section 7. Initial experiments with use of *P*-logic to prove properties of small but interesting Haskell functions are reported in Section 8. The paper concludes in Section 9 with a discussion of related work and preliminary conclusions of the research.

2 A logic for a non-strict programming language

In the lazy-evaluation semantics of Haskell, a term does not necessarily denote a value, as a term does not need be normalized until its value is demanded. Thus a logic expressing properties of Haskell terms cannot be interpreted only over values; its interpretation must also accommodate latent computations. There are three possible answers to the proposition that a term, t, exhibits a property, P; namely True, False or Maybe. The answer "Maybe" is appropriate whenever t can be interpreted by a computation that has not reached a normal form. Because of Haskell's lazy evaluation rules, such terms abound.

The atomic propositional forms of *P*-logic either assert term equality, $t_1 \mathrel{===} t_2$, or that a term has a property expressed as a unary predicate, $t ::: P$, or more generally, that a k-ary predicate applies to k terms. The symbol (:::) expresses a (unary) property assertion, in analogy to the symbol (::) that expresses a typing assertion in Haskell.

Informally, we intend that an assertion $t ::: P$ should be true in a given context if evaluating t by Haskell's operational semantics results in a normal form that manifestly satisfies the property P. But what if evaluation of t is not necessarily demanded in the program context in which the assertion occurs? Because of Haskell's non-strict evaluation semantics, two notions of satisfaction of a (unary) predicate are sensible.

We say that a predicate, P, is *weakly* satisfied by an expression t if the denotation of t belongs to the set defined by the interpretation of P. It is *strongly* satisfied if in addition, the denotation of t is not the bottom element in its type domain. In *P*-logic, the strength associated with an assertion is determined primarily from the syntactic context in which it occurs. However, an otherwise weak assertion context can be explicitly strengthened by prefixing the symbol (\$) to a predicate formula. This constrains an interpretation of the predicate to be False of a computationally undefined expression.

There are four principal ways that compound predicate formulas are formed in *P*-logic.

- The connectives of the underlying propositional logic are "lifted" to work as connectives on predicates. For example, the conjunction connective, ($\wedge$) is lifted to a predicate connective by the definition $x ::: (P \wedge Q) =_{def} (x ::: P \wedge x ::: Q)$.
- Constructors of datatypes declared in a Haskell program text 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 *List a* into a new expression $(h : t)$ of type *List a*. But in the context of a formula, the same constructor combines a predicate P and a predicate Q into a new predicate formula, $(P : Q)$. This formula is satisfied by a Haskell expression that normalizes to a term of the form $(h : t)$ and whose component expressions weakly satisfy the assertions $h ::: P$ and $t ::: Q$. The default mode of interpretation of the component predicates is weak because the semantics of the data constructor allow bottom elements as its arguments.

- The "arrow" predicate constructor is used to compose formulas that express properties of function-typed expressions. A formula $P \to Q$ is satisfied by a function if when the function is applied to an argument that weakly satisfies P, the resulting application strongly satisfies Q.
- A least or greatest fixpoint constructor may bind a predicate variable in a prefix appended to a predicate formula. The fixpoint constructors are the μ and ν binders of the mu-calculus, but are written as Lfp and Gfp in formulas of *P*-logic.

2.1 The modality of Haskell terms

Taken together, the connectives, predicate constructors and fixpoint binders allow detailed properties of Haskell expressions to be formulated in *P*-logic. For example, a predicate asserting that an expression of type *List a* denotes a finite list can be defined as

$$\mathsf{property}\ \mathit{Finite\_list} = \mathsf{Lfp}\,X.\,(\,\texttt{[]}\ \vee\ (\mathsf{Univ} : \$X))$$

in which Univ is the universal predicate, satisfied by every term.

The body of the formula *Finite_list* asserts the disjunction of two constructor formulas. The first, a lifting of the data constructor `[]`, is satisfied by an expression denoting the nil list. The second, a lifting of the list constructor, is satisfied by a term denoting a constructed list whose head satisfies the universal property (or is undefined) and whose tail is defined and satisfies the *Finite_list* property.

To characterize lists for which every listed element satisfies a predicate parameter, Q, declare

$$\mathsf{property}\ \mathit{All\_elts}\ Q =_{\mathit{def}} \mathsf{Gfp}X.(\,\texttt{[]}\ \vee\ (Q : \$X))$$

The property *All_elts*, defined with a greatest fixpoint binding, includes both finite and infinite lists. The property definition differs from that of *Finite_list* in that Gfp, rather than Lfp, is the binding operator in the formula.

Detailed properties of the term structures that model values of Haskell datatypes are directly expressible in the modalities of *P*-logic. It is difficult to overemphasize the importance of achieving this degree of expressiveness in a verification logic.

3 The syntax of P-logic

The syntax of *P*-logic is an important issue, as property declarations and assertions are intended to be embedded in the text of a Haskell program, yet the logic also uses Haskell as its term language. It is important that the syntax provides visual distinction between a property assertion and an expression such as a function application. Such a distinction can help to avoid semantic confusion on the part of a reader.

3.1 Declarations, properties and assertions

There are certain contexts in a Haskell program where equational declarations are expected. Declarations appear at the top level in a module, or embedded in an expression following the keyword `where` or between the keywords `let` and `in`. Property declarations and assertions are allowed in exactly these contexts in a property-annotated program.

A property declaration is prefaced by the keyword `property` and an assertion is prefaced by the keyword `assert`, to provide visual distinction from Haskell expressions as well as to allow a parser to recognize formulas as distinct from expressions. In a text segment prefaced by one of these keywords, certain symbols have special roles, different from their customary roles as operator symbols in Haskell. In ordinary Haskell program text, the symbols revert to their customary roles. This change of role for lexical symbols is necessary because the Haskell language design has already assigned a role in a name space to virtually every printable Ascii character and short sequence of characters, leaving little room for the creation of new lexemes unless they occur in a prescribed context.

3.2 Identifiers and reserved symbols

The syntax of *P*-logic reserves a small number of identifiers for use in formulas, in effect "stealing" these identifiers from the name spaces of operators, variables and data constructors.

- `property` and `assert` are reserved keywords.
- The symbols "`:::`", "`===`", "`/\`", "`\/`" and "`\!`" are reserved for use as assertion-forming operators and should not be used as data constructor or operator symbols in a Haskell program.
- The identifiers *All* and *Exist* are reserved for use as quantifiers in assertions and should not be used as data constructors in a Haskell program.
- The identifiers `Univ` and `UnDef` are reserved for predicate constants and should not be used as data constructors in a Haskell program.

The operator symbols '!' and '$' and the identifiers `Lfp` and `Gfp` also have special uses in the syntax of formulas. They are not reserved from use as operators in Haskell expressions or data declarations, however.

3.2.1 Name spaces

Predicate constructors and propositions can be named in `property` and `assert` declarations, respectively. The name space occupied by predicate and proposition names (as well as predicate variables) coincides with the name space of data constructors in Haskell. Identifiers in this name space have their initial letter capitalized.

The use of a name in this space can be resolved syntactically in any annotated program fragment that provides sufficient context, i.e. a constructor name denotes a predicate constructor when it occurs within a property or assertion declaration or a set comprehension formula, but denotes a data constructor when used in an expression context.

3.3 Predicates and predicate constructors

A predicate in *P*-logic is either

- a unary predicate formula, which characterizes a set of Haskell terms by specifying properties of their semantic interpretation, or
- the binary equality predicate.

Formulas and Declarations

assertDecl	::=	*Assert* Identifier assertion	a declaration names an assertion
assertion	::=	*PropConst* Identifier	a proposition constant
	|	*PropVar* Identifier	a proposition variable
	|	*Has* term pred	asserts a property of a term
	|	*Equal* term term	asserts equality of two terms
	|	*k-aryProp* Identifier term*	asserts a property of k terms
	|	*Conj* assertion*	conjunction of propositions
	|	*Disj* assertion$^*b$	disjunction of propositions
	|	*Neg* assertion	negation of a proposition
	|	*Quant* qvar* assertion	assertion with a quantification prefix
qvar	::=	*All* identifier type	universally quantified term variable
	|	*Exists* identifier type	existentially quantified term variable
pred-decl	::=	*PredDecl* pred-pat pred	predicate declaration
pred-pat	::=	*Pred-abs* Identifier (identifier | Identifier)*	predicate pattern
pred	::=	*PredConst* Identifier	a predicate constant
	|	*Arrow* pred pred	formula of a function property
	|	*Pred-disj* pred*	disjunction of formulas
	|	*Pred-conj* pred*	conjunction of formulas
	|	*Pred-neg* pred	negation of a formula
	|	*PredInst* Identifier (term | pred)*	predicate instance
	|	*Lfp-pred* Identifier pred	l.f.p. predicate formula
	|	*Gfp-pred* Identifier pred	g.f.p. predicate formula
	|	*Pred-ref* Identifier	named predicate or constructor
	|	*Pred-var* Identifier	predicate variable
	|	*Lifted-section* term	lifted section
	|	*Pred-section* pred Nat term	predicate section (pred must be 2-ary)
	|	*Strong* pred	strengthened predicate
	|	*Comprehension* (identifier type)* pred	set comprehension formula
	|	*TypedPred* pred type	typed predicate

Rules and Axioms

rule	::=	*Rule* sequent* sequent	
	|	*Axiom* sequent	
sequent	::=	*Entails* (assertion | assumption)* assertion*	
assumption	::=	*DefEqual* variable term	

where "term" is a Haskell term, possibly containing variables, and the term in a "*Lifted-section*" property is a Haskell section, such as (< 1). Use of the word "identifier" both capitalized and uncapitalized is not accidental. It distinguishes meta-variables of the abstract syntax that range over formulas and terms, respectively.

Figure 1: Abstract syntax of forumlas

NT		**produces**	**AST**
assertion	$\rightarrow$	`assert` [*Ident* `=`] *proposition*	*AssertDecl Ident* $\overline{\mathit{proposition}}$ | $\overline{\mathit{proposition}}$
proposition	$\rightarrow$	*qvar** *uprop*	*Quant qvar** $\overline{\mathit{uprop}}$ | $\overline{\mathit{uprop}}$
qvar	$\rightarrow$	*quantifier ident* [‘`::`’ *type*] ‘`.`’	$\overline{\mathit{quantifier}}$ ident type
quantifier	$\rightarrow$	`All`	*All*
	|	`Exist`	*Exist*
uprop	$\rightarrow$	*aprop*	$\overline{\mathit{aprop}}$
	|	‘$\neg$’ *aprop*	*Neg* $\overline{\mathit{aprop}}$
	|	*aprop* ‘$\vee$’ *uprop*	*Disj* $\overline{\mathit{aprop}}$, $\overline{\mathit{uprop}}$
	|	*aprop* ‘$\wedge$’ *uprop*	*Conj* $\overline{\mathit{aprop}}$, $\overline{\mathit{uprop}}$
aprop	$\rightarrow$	Univ	*Prop-const* Univ
	|	UnDef	*Prop-const* UnDef
	|	$EXPR_1$ ‘`===`’ $EXPR_2$	*Equal* $EXPR_1$ $EXPR_2$
	|	*EXPR* ‘`:::`’ *formula*	*Has EXPR* $\overline{\mathit{formula}}$
	|	*Ident* $EXPR_1 \cdots EXPR_k$	*k-aryProp Ident* $[EXPR_1; \ldots; EXPR_k]$
property_decl	$\rightarrow$	`property` *prop-pat* ‘`=`’ *formula*	*PropDecl* $\overline{\mathit{prop\text{-}pat}}$ $\overline{\mathit{formula}}$
prop-pat	$\rightarrow$	*Ident*	*Pred-abs Ident* [ ]
	|	*Ident* [*ident* | *Ident*]*	*Pred-abs Ident* [(*ident* | *Ident*)*]
formula	$\rightarrow$	*aformula*	$\overline{\mathit{aformula}}$
	|	‘$\neg$’ *aformula*	*Pred-neg* $\overline{\mathit{aformula}}$
	|	*aformula* ‘$\vee$’ *formula*	*Pred-disj* [$\overline{\mathit{aformula}}$; $\overline{\mathit{formula}}$]
	|	*aformula* ‘$\wedge$’ *formula*	*Pred-conj* [$\overline{\mathit{aformula}}$; $\overline{\mathit{formula}}$]
	|	*aformula* ‘`->`’ *aformula*	*Arrow* $\overline{\mathit{aformula}}$ $\overline{\mathit{formula}}$
	|	*formula* ‘`:`’ *formula*	*PredInst* “*Cons*” [$\overline{\mathit{formula}}$; $\overline{\mathit{formula}}$]
	|	`Lfp` *Ident* ‘`.`’ *formula*	*Lfp-pred Ident* $\overline{\mathit{formula}}$
	|	`Gfp` *Ident* ‘`.`’ *formula*	*Gfp-pred Ident* $\overline{\mathit{formula}}$
	|	*Ident* [*AEXPR* | *formula*]*	*PredInst Ident* [(*AEXPR* | $\overline{\mathit{formula}}$)*]
		(predicate constructor applications may not be infixed)	
aformula	$\rightarrow$	*Ident*	*Pred-ref Ident*
	|	‘`[]`’	*PredInst* “*Nil*” [ ]
	|	‘`!`’ *AEXPR*	*Lifted-section AEXPR*
		(an *AEXPR* is a single token or a parenthesized *EXPR*)	
	|	(*Ident EXPR*)	*Pred-section Ident* 1 *EXPR*
	|	(‘*Ident*‘ *EXPR*)	*Pred-section Ident* 2 *EXPR*
	|	‘`$`’ *aformula*	*Strong* $\overline{\mathit{aformula}}$
	|	‘`{|`’ *Ident* [‘`::`’ *type*] ‘`|`’ *proposition* ‘`|}`’	*Comprehension Ident* $\overline{\mathit{type}}$ $\overline{\mathit{proposition}}$
	|	’`(`’ *formula* ‘`)`’	$\overline{\mathit{aformula}}$

An overlined syntactic category stands for the translation of the phrase it represents to an abstract syntax tree.

Figure 2: Concrete syntax of formulas

The basic forms of unary predicate formulas are

- predicate constants, `Univ`, `UnDef`
- (set) comprehensions
- patterned predicates
- lifted section expressions;
  a section expression whose range type is *Bool* can be designated as lifted by a prefix '!', as for example, `!(>0)`.

The '!' symbol is preempted for use as a prefix operator in formulas. This does not conflict with its use as a reserved symbol in datatype declarations nor as an operator symbol in Haskell expression.

A lifted section expression is restricted with respect to the scope of identifiers that may appear within it. Identifiers must either be in the scope of a Haskell language-specified definition, or must be logical variables. A logical variable is a variable symbol bound in a quantifier prefix or introduced in a set comprehension formula.

3.3.1 Multi-place predicates

A multi-place predicate can be defined by a set comprehension in which more than one term variable is bound. No special syntax is provided for multi-place predicates other than the equality predicate.

Note that a multi-place predicate is not the same thing as a predicate constructor that is explicitly abstracted over term variables. A constructor abstracted over term variables may be instantiated by applying it to a Haskell term composed only of constants, predefined operators and logical variables. A multi-place, or k-ary predicate specifies a relation among unconstrained terms whose values may range over their respective types.

3.3.2 Predicate sections

Sectioning can be used with a binary predicate to bind either the first or second argument, resulting in a unary predicate. For example, the equality predicate can be sectioned with a term t, to obtaining a unary predicate (t `===`) or (`===` t), as equality is symmetric. Just as with lifted section predicates, the term argument given in a predicate section is restricted to contain only constants, predefined operators and logical variables.

3.3.3 Compound predicate formulas

Compound predicates are formed with least and greatest fixed-point operations on predicates, or with the "lifted" propositional connectives `-/`, `/\` and `\/`, or by applying a predicate constructor to arguments of suitable kind and type.

There are three forms of predicate constructors,

1. The unary, predicate-strengthening operator, '$', mandates a strong interpretation of a formula given as its argument even if the strengthened formula is embedded in an otherwise weak position.
2. a data constructor of a Haskell datatype can be "lifted" to a predicate constructor. A lifted data constructor, $C^{(k)}$, of arity $k \geq 0$ constructs a pattern predicate when it occurs in a predicate

formula. A data constructor typed as $C^{(k)} :: \tau_1 \to \cdots \to \tau_k \to \tau$ can also occur in a predicate expression, where it assumes the predicate type $\mathsf{Pred}(\tau_1) \to \cdots \to \mathsf{Pred}(\tau_k) \to \mathsf{Pred}(\tau)$.

To construct a predicate, a k-ary constructor is applied to k unary predicate formulas. A constructor appearing in a predicate formula requires no syntactic annotation to distinguish it from a data constructor, as the context of its occurrence resolves the potential ambiguity. Such contextual interpretations already exist in Haskell to allow constructors to occur either in patterns or in expressions. Here we take advantage of contextual resolution to allow predicate patterns to be formed with data constructors.

3. An identifier may be explicitly defined as a predicate constructor in a predicate declaration following the keyword `property`. It is written as an equation whose left-hand side is a simple applicative pattern and whose right-hand side is a predicate formula. The formula on the right-hand side is abstracted with respect to the variables bound in the applicative pattern.

   Notice that the argument of a unary predicate cannot be abstracted as a variable in a left-hand side pattern, for then the right-hand side would have the type *Prop*, the type of an assertion, whereas it must have the type of a predicate over a type of terms, τ, namely $\mathsf{Pred}(\tau)$.

Note that the symbol '\$' is *not* used as a right-associative infix operator for applications of predicate constructors to their arguments, as it is in Haskell expression syntax.

3.3.4 Property assertions

A property assertion consists of a propositional formula prefixed by quantifier clauses that bind logical term variables in the formula. The variables introduced in a set comprehension are also logical variables, implicitly quantified over their types.

A basic propositional formula is either

1. a unary predicate formula applied to a Haskell expression, or
2. two expressions related by the infix equality predicate, `(===)`, or
3. the application of a predicate formula typed as $\mathsf{Pred}\ \tau_1 \cdots \tau_k$ to k term arguments of types $\tau_1 \cdots \tau_k$, respectively. A predicate reference naming a binary predicate may be infixed by enclosing it in backquotes.

Assertions prefaced by the keyword `assert` may appear (i) at the top level in a program text, (ii) interleaved among a sequence of definitions in a let expression or a where clause, or (iii) in the body of a set comprehension expression. Assertions at the top level may (optionally) be named.

A unary property assertion relates a term of type τ to a predicate formula of complementary type, $\mathsf{Pred}(\tau)$. An assertion stands for itself; there are no evaluation rules by which to reduce or eliminate it. To assert that a term t has the property characterized by a predicate, P, we write $t ::: P$. Since a predicate specifies a set of values, an assertion $t ::: P$ can also be read as "t is a member of the set P".

The application of a predicate to a term is given a different syntax than is function application because it has a different semantics. Function application, or the application of a predicate constructor to abstracted arguments, has a reduction semantics. This semantics can be described operationally as substitution of argument expressions into the body of a function. There is no such operational description of property satisfaction. A predicated assertion may be logically equivalent to other assertions, as deduced from the inference rules of the logic, but an assertion is never reduced.

3.4 Type-indexed predicates

Predicate formulas and propositions in P-logic are strongly typed, as are expressions in Haskell. The Haskell type system is extended with four additional kinds,

$$\mathit{Kind} ::= \mathit{Type} \mid \mathit{Prop} \mid \mathit{Pred} \mid \mathit{Maybe} \mid \mathit{Maybe\_Pred}$$

which distinguish the following:

- the types of Haskell expressions, $\tau :: \mathit{Type}$;
- the type of propositions (or assertions) $\mathsf{Prop} :: \mathit{Prop}$;
- the types of predicate formulas, $\mathsf{Pred}\ \tau :: \mathit{Pred}$ where $\tau :: \mathit{Type}$;
- the types of match clauses, $\mathsf{Maybe}\ \tau :: \mathit{Maybe}$
- the types of predicates over match clauses,$\mathsf{Pred}\ (\mathsf{Maybe}\ \tau) :: \mathit{Maybe\_Pred}$.

Predicate formulas in P-logic may be indexed by types that mirror Haskell types. Type-indexing is necessary to resolve instances of overloading when lifted operators occur in formulas, as for instance, in lifted section expressions. It is needed to express rules that characterize properties of standard algebras, such as the various arithmetic algebras for which operations are implemented in Haskell.

The type index of a formula specifies the type of terms that may satisfy the formula. In Ascii text, a predicate type appears as a Haskell type expression bracketed by $\mathsf{Pred}(\cdots)$, as in $!(> 0) :: \mathsf{Pred}(\mathsf{Integer})$. Explicit typing of predicates is not required when a type can be inferred. The type index associated with an assertion of equality between a pair of Haskell expressions is the (common) type of both expressions.

3.4.1 Typing rules for predicate formulas

A type-indexed predicate formula can be judged to be well-typed (or not) relative to a typing environment derived from the context provided by the Haskell program in which the formula occurs. This typing environment extends the typing environment of the program, augmenting the type bindings of program variables with typings for predicate variables that are in scope. We shall express the rules for typing formulas in terms of sequents, in which Γ_T stands for a typing environment and $(\vdash_T)$ stands for the typing consequence relation. Typing rules are given in Figure 3.

3.4.2 Class qualifications and instance types

Haskell supports operator overloading through its class mechanism. A class declaration gives the signatures of operators that must be defined in every type instance of the class. Resolution of overloading in a Haskell program is not always possible through static type inference alone. Dynamic resolution is provided by a dictionary-passing mechanism supported by a Haskell implementation.

Property verification relies upon type-indexed predicates rather than the dynamic, dictionary-passing mechanism to resolve overloading. The typing context of an operator occurrence in a property assertion is determined by a type-indexed predicate. This typing context must be compatible with the types statically inferred from the Haskell program but it may refine them.

A declaration whose type ranges over the instances of a class may be accompanied by a set of type-indexed assertions, each asserting properties that may differ. For example, the operator $(+)$, which has the type $\mathit{Num}\ a \rightarrow a \rightarrow a$ in the Haskell standard prelude, might have associated with it property assertions such as $(+) ::: \mathit{Assoc} :: \mathsf{Pred}(\mathit{Integer})$ and $(+) ::: \mathit{Assoc} :: \mathsf{Pred}(\mathit{Int})$. This property of $(+)$ need not

The universal predicate may be indexed with any type.

(*Universal-type*) $\tau :: \mathit{Type} \vdash_T \mathsf{Univ} :: \mathsf{Pred}(\tau)$

Typing rules for list constructor congruences.

(*Nil-list-type*) $\tau :: \mathit{Type} \vdash_T \texttt{[]} :: \mathsf{Pred}([\tau])$

(*Cons-list-type*) $$\frac{\Gamma_T \vdash_T P :: \mathsf{Pred}(\tau) \quad \Gamma_T \vdash_T Q :: \mathsf{Pred}([\tau])}{\Gamma_T \vdash_T (P : Q) :: \mathsf{Pred}([\tau])}$$

Typing rules for other datatype constructor congruences.

(*Congruence-type*) $$\frac{\Gamma_T \vdash_T P_1 :: \mathsf{Pred}(\tau_1) \quad \cdots \quad \Gamma_T \vdash_T P_k :: \mathsf{Pred}(\tau_k)}{\Gamma_T,\, C^{(k)} :: \tau_1 \to \cdots \to \tau_k \to \tau \vdash_T C^{(k)}\ P_1 \cdots P_k) :: \mathsf{Pred}(\tau)}$$

Typing rule for arrow-typed predicates.

(*Arrow-type*) $$\frac{\Gamma_T \vdash_T P :: \mathsf{Pred}(\tau_1) \quad \cdots \quad \Gamma_T \vdash_T Q :: \mathsf{Pred}(\tau_2)}{\Gamma_T \vdash_T P \to Q :: \mathsf{Pred}(\tau_1 \to \tau_2)}$$

Typing rule for set comprehensions.

(*Comprehension-type*) $$\frac{\Gamma_T,\, x_1 :: \tau_1, \ldots, x_k :: \tau_k \vdash_T R :: \mathsf{Prop}}{\Gamma_T \vdash_T \texttt{[|}\, x_1, \ldots, x_k \ \mid\ R\, \texttt{|]} :: \mathsf{Pred}\, \tau_1 \cdots \tau_k}$$

Typing rules for formulas with logical connectives.

(*Negation-type*) $$\frac{\Gamma_T \vdash_T P :: \mathsf{Pred}(\tau)}{\Gamma_T \vdash_T \neg P :: \mathsf{Pred}(\tau)}$$

(*Conjunction-type*) $$\frac{\Gamma_T \vdash_T P :: \mathsf{Pred}(\tau) \quad \Gamma_T \vdash_T Q :: \mathsf{Pred}(\tau)}{\Gamma_T \vdash_T P \wedge Q :: \mathsf{Pred}(\tau)}$$

(*Disjunction-type*) $$\frac{\Gamma_T \vdash_T P :: \mathsf{Pred}(\tau) \quad \Gamma_T \vdash_T Q :: \mathsf{Pred}(\tau)}{\Gamma_T \vdash_T P \vee Q :: \mathsf{Pred}(\tau)}$$

(*Strengthened-type*) $$\frac{\Gamma_T \vdash_T P :: \mathsf{Pred}(\tau)}{\Gamma_T \vdash_T \$P :: \mathsf{Pred}(\tau)}$$

(*Lfp-type*) $$\frac{\Gamma_T,\, X :: \mathsf{Pred}(\tau) \vdash_T H :: \mathsf{Pred}(\tau)}{\Gamma_T \vdash_T \mathsf{Lfp}\, X.H :: \mathsf{Pred}(\tau)}$$

(*Gfp-type*) $$\frac{\Gamma_T,\, X :: \mathsf{Pred}(\tau) \vdash_T H :: \mathsf{Pred}(\tau)}{\Gamma_T \vdash_T \mathsf{Gfp}\, X.H :: \mathsf{Pred}(\tau)}$$

Typing rules for well-typed propositions.

(*Pred-formula-type*) $$\frac{\Gamma_T \vdash_T P :: \mathsf{Pred}(\tau) \quad \Gamma_T \vdash_T t :: \tau}{\Gamma_T \vdash_T (t ::: P) :: \mathsf{Prop}}$$

(*Equality-type*) $$\frac{\Gamma_T \vdash_T t_1 :: \tau \quad \cdots \quad \Gamma_T \vdash_T t_2 :: \tau}{\Gamma_T \vdash_T (t_1 \mathrel{===} t_2) :: \mathsf{Prop}}$$

(*k-ary-formula-type*) $$\frac{\Gamma_T \vdash_T P :: \mathsf{Pred}\, \tau_1 \cdots \tau_k, \ \ t_1 :: \tau_1, \ \cdots \ t_k :: \tau_k}{\Gamma_T \vdash_T P\, t_1 \cdots t_k :: \mathsf{Prop}} \qquad (k \geq 2)$$

Figure 3: Typing rules for formulas

be asserted at other instances of the class *Num*. The typing context of an occurrence of (+) in a subsequent assertion will determine which, if any, of the properties associated with the operator belong to the logical context of that particular occurrence.

A formula containing operator symbols can also be overloaded, which allows generic properties of the derived instances of an overloaded operator to be asserted. Generic properties may be associated with a derived operator defined in a prelude or a library module. Generic properties of a derived operator can be assumed in a verification context in which the operator is in scope. Naturally, such an assertion will not bind to program-specified instances of an operator symbol.

Many predefined classes contain declarations of infix operators, which may be embedded in lifted-section predicates. A typical typing rule for a section predicate is given below. The typing by itself cannot determine whether the predicate is associated with a derived instance of the operator or with a program-defined instance.

$$\boxed{(<\textit{-section-type}) \qquad \Gamma_T,\, a :: \mathsf{Ord}\ \tau\ \texttt{=>}\ \tau \vdash_T\ !(<a) :: \mathsf{Pred}(\tau)}$$

3.5 Examples

```
assert mapFunctor =
       All f :: b -> c.
         All g :: a -> b.
           map (f . g) === map f . map g

assert CompositionIsAssociative =
       All f :: c -> d.
         All g :: b -> c.
           All h :: a -> b.
             f . (g . h) === (f . g) . h

-- A property specified by set abstraction

property Assoc = {| op :: a -> a -> a |
                       All x :: a.
                         All y :: a.
                           All z :: a.
                             op (op x y) z === op x (op y z) |}
                 :: Pred(Int -> Int -> Int)

assert (+) ::: Assoc

-- Well-ordering: A predicate constructor declared by explicit abstraction

property WO (p :: a -> Bool) (b :: a -> a) =
      Lfp X . {| x :: a | (p x === True) \/ (b x ::: X) |}

property WO_Int = WO (== 0) (subtract 1) :: Pred(Int)

assert factorial ::: WO_Int -> WO_Int
```

```
-- A comprehension defining a binary predicate; uses predicate sectioning

property IsInserted =
      Gfp X. [| n::Int, xs::[Int] | xs:::($(== n):True) \/ ($(< n):($X n)) |]

assert All n :: Int.
         All xs :: [Int].
           n `IsInserted` (insert n xs)
```

b

4 Rules of P-logic

Deductions in P-logic are formally carried out in a sequent calculus. The rules of P-logic fall into two sets:

> **Structural rules** support the inference of new sequent formulas from existing ones, preserving validity independently of the structure of particular assertions;
>
> **Content rules** support the inference of sequents valid with respect to a standard interpretation of terms, modal operators and predicate constructors.

4.1 Structural rules

The trivial sequent, $A \vdash A$, is an axiom of P-logic.

Because of the strong and weak interpretations of predicates in P-logic, the negation of a strong assertion is not weak, nor vice-versa. Consequently, the swap rule, which is valid in classical sequent calculii, has two forms in P-logic.

$$(\textit{Strong-swap}) \quad \begin{array}{c} \Gamma,\, \neg P^{(k)}\, t_1 \cdots t_k \vdash \Xi \\ \hline\hline \Gamma \vdash \$P^{(k)}\, t_1 \cdots t_k,\, \Xi \end{array} \qquad\qquad (\textit{Weak-swap}) \quad \begin{array}{c} \Gamma,\, \$\neg P^{(k)}\, t_1 \cdots t_k \vdash \Xi \\ \hline\hline \Gamma \vdash P^{(k)}\, t_1 \cdots t_k,\, \Xi \end{array}$$

where the doubled lines indicate that a rule is invertible.

Additional rules specify relations among assumptions and conclusions of a sequent:

$$(\textit{Left symmetry}) \quad \frac{\Gamma,\, A,\, B \vdash \Xi}{\Gamma,\, B,\, A \vdash \Xi} \qquad\qquad (\textit{Right symmetry}) \quad \frac{\Gamma \vdash A,\, B,\, \Xi}{\Gamma \vdash B,\, A,\, \Xi}$$

$$(\textit{Left conj}) \quad \frac{\Gamma,\, A,\, B \vdash \Xi}{\Gamma,\, (A \wedge B) \vdash \Xi} \qquad\qquad (\textit{Right disj}) \quad \frac{\Gamma \vdash A,\, B,\, \Xi}{\Gamma \vdash (A \vee B),\, \Xi}$$

$$(\textit{Left strengthen}) \frac{\Gamma,\, P^{(k)}\, t_1 \cdots t_k \vdash \Xi}{\Gamma,\, \$P^{(k)}\, t_1 \cdots t_k \vdash \Xi} \qquad\qquad (\textit{Right weaken}) \frac{\Gamma \vdash \$P^{(k)}\, t_1 \cdots t_k,\, \Xi}{\Gamma \vdash P^{(k)}\, t_1 \cdots t_k,\, \Xi}$$

$$(\textit{Strengthen assumptions}) \quad \frac{\Gamma \vdash \Xi}{\Gamma,\, A \vdash \Xi} \qquad\qquad (\textit{Weaken conclusions}) \quad \frac{\Gamma \vdash \Xi}{\Gamma \vdash A,\, \Xi}$$

$$(\textit{Left contraction}) \quad \frac{\Gamma,\, A \vdash \Xi \qquad \Gamma \vdash A}{\Gamma \vdash \Xi}$$

4.1.1 Propositional connectives are lifted to predicate connectives

The propositional connectives $(\wedge)$ and $(\vee)$ are also defined as unary predicate connectives, by the following equivalences. Negation is lifted to a connective for predicates of arbitrary arity.

$$\boxed{\begin{array}{ll} \textit{Lift-conj} & t ::: (P \wedge Q) \equiv_{def} (t ::: P) \wedge (t ::: Q) \\ \textit{Lift-disj} & t ::: (P \vee Q) \equiv_{def} (t ::: P) \vee (t ::: Q) \\ \textit{Lift-negate} & (\neg P^{(k)})\, t_1 \cdots t_k \equiv_{def} \neg(P^{(k)}\, t_1 \cdots t_k) \end{array}}$$

4.1.2 Equality

We assume the reflexivity, symmetry and transivity of the equality predicate, (`===`). In addition, the following axiom states that equality transfers other properties.

$$t ::: P,\, t \mathtt{===} s \vdash s ::: P \tag{1}$$

4.2 Content rules

4.2.1 Pair construction and projections

$$\frac{\Gamma \vdash x ::: P \qquad \Gamma \vdash y ::: Q}{\Gamma \vdash x, y \;:::\; \$(P,\, Q)} \tag{2}$$

$$\frac{\Gamma \vdash p ::: \$(P,\, \mathsf{Univ})}{\Gamma \vdash \mathsf{fst}\, p \;:::\; P} \qquad\qquad \frac{\Gamma \vdash p ::: \$(\mathsf{Univ},\, Q)}{\Gamma \vdash \mathsf{snd}\, p \;:::\; Q} \tag{3}$$

The predicate asserting a pair construction in the antecedent for the rules for fst and snd projections is explicitly strengthened, as pairs in Haskell are built with an explicit, language-defined constructor. Thus normalization of a product-typed expression may be independent of the normalization of either component.

4.2.2 Abstraction

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.

$$\frac{\Gamma,\, x ::: P \vdash t ::: Q}{\Gamma \vdash (\lambda x \mathtt{->} t) \;:::\; \$(P \to Q)} \tag{4}$$

To apply this rule to an abstraction whose body, t, is strict in the abstracted variable, it will be necessary to furnish an explicitly strengthened predicate, $\$P'$, in order to discharge the proof obligation of the antecedent. In that case, the consequent property of the strict abstraction will be $\$(\$P' \to Q)$.

4.2.3 Application

$$\frac{\Gamma \vdash t_1 ::: \$(P \to Q) \qquad \Gamma \vdash t_2 ::: P}{\Gamma \vdash t_1\, t_2 \;:::\; Q} \tag{5}$$

A coinductive rule allows a property of a function-typed term to be deduced from a property of its application to a variable:

$$\frac{\Gamma,\, x ::: P \vdash f\, x ::: Q}{\Gamma \vdash f ::: \$(P \to Q)} \qquad \text{where } x \text{ is a variable} \tag{6}$$

4.2.4 Constructor application

When a k-place data constructor, $C^{(k)}$, is defined in a datatype declaration

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

an inductive rule schema for a saturated application of the application is

$$\frac{\Gamma \vdash t_1 ::: P_1 \ \cdots \ \Gamma \vdash t_k ::: P_k}{\Gamma \vdash C^{(k)}\ t_1 \ldots t_k ::: \$C^{(k)}\ P_1 \ldots\ P_k} \qquad (k \geq 0) \tag{7}$$

The data constructor is implicitly lifted to serve as a predicate constructor. The predicate formula $C^{(k)}\ P_1 \ldots\ P_k$ is satisfied by a well-defined term of the form $C^{(k)}\ t_1\ \ldots\ t_k$ whenever each of the t_j (weakly) satisfies the corresponding predicate P_j. This is a term congruence, which enriches the formula language.
This rule, which applies only to an explicitly constructed term, will be generalized to account for pattern-matching constructs.

A restricted form of the rule applies when the constructor, $C^{(k)}$, has been declared to be strict in one or more of its arguments, as in

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

Then

$$\frac{\Gamma \vdash t_1 ::: P_1 \ \cdots \ \Gamma \vdash t_j ::: \$P_j \ \cdots \ \Gamma \vdash t_k ::: P_k}{\Gamma \vdash C^{(k)}\ t_1 \ldots t_j \ldots t_k ::: \$C^{(k)}\ P_1 \ldots \$P_j \ldots P_k} \qquad (1 \leq j \leq k) \tag{8}$$

4.3 Pattern matching

Haskell allows patterns to be used in abstractions, local (let) definitions and case instances. When an attempt is made to match a pattern to a term, the match either succeeds and binds subterms to local variables occurring in the pattern, or it fails.

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

4.3.1 Coercions from the Maybe kind

Two rules relate a property of kind *Maybe_Pred* to a property of kind *Pred.*

$$t\ :::\ \mathsf{Just}(P) \vdash t\ :::\ P \tag{9}$$

$$t\ :::\ \mathsf{Nothing} \vdash t\ :::\ \mathsf{Univ} \tag{10}$$

These rules allow a property of an application of a patterned abstraction to be propagated to a context that expects a property of kind *Pred.* In a context in which the pattern match in the application can be shown to fail, the resulting property, Univ, provides no specific information.

4.3.2 Pattern predicates

Patterns are nested terms formed with variables, the wildcard pattern symbol, "_" and saturated applications of data constructors to arguments that are patterns. Every occurrence of a variable in a pattern is deemed to be a binding occurrence, hence, no variable can have a repeated occurrence within a given pattern.

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

Definition 4.3.2.1: The *skeleton* of a pattern is the formula defined inductively as follows:

$$\begin{array}{ll} \mathit{skel}\; x_i = X_i & \text{where } x_i \text{ is a variable} \\ \mathit{skel}\; \_ = \mathsf{Univ} & \\ \mathit{skel}\; (C^{(k)}\; p_1 \cdots p_n) = \$C^{(k)}\; (\mathit{skel}\, p_1) \cdots (\mathit{skel}\, p_n) & \\ \mathit{skel}\; (\sim C^{(k)}\; p_1 \cdots p_n) = C^{(k)}\; (\mathit{skel}\, p_1) \cdots (\mathit{skel}\, p_n) & \end{array}$$

Note that every constructor in a pattern is explicitly strengthened when it is lifted to a predicate in a skeleton, unless the constructor is prefixed with $(\sim)$ to designate an occurrence that is is not to be strengthened. (In Haskell, a $(\sim)$-prefixed pattern is called "irrefutable".)

Definition 4.3.2.2: The *fringe* of a pattern is the list of (distinct) variables defined inductively as follows:

$$\begin{array}{ll} \mathit{frg}\; x = [x] & \text{where } x \text{ is a variable} \\ \mathit{frg}\; \_ = [\,] & \\ \mathit{frg}\; C^{(k)}\; p_1 \cdots p_n = \mathit{frg}\, p_1 \mathbin{+\!+} \cdots \mathbin{+\!+} \mathit{frg}\, p_n & \end{array}$$

If the fringe of a pattern consists of n variables, the skeleton of the pattern is a formula containing n free predicate variables. Closing this formula by abstracting over the predicate variables yields a *pattern predicate constructor*.

Definition 4.3.2.3: $\pi(p) = [X_1, \ldots, X_n]\;\; \mathit{skel}(p)$ is the predicate constructor derived from a pattern, where $[x_1, \ldots, x_n] = \mathit{frg}(p)$.
□

The above definition justifies the notation $\pi(p)\, P_1 \cdots P_n$ to represent the pattern predicate derived from a pattern, p, by substituting a predicate, P_i, for each of the term variables x_i in its fringe.

If a pattern, p, satisfies the typing judgment

$$x_1 :: \tau_1, \ldots, x_n :: \tau_n \vdash_T p :: \tau \qquad \text{where } [x_1, \ldots, x_n] = \mathit{frg}(p)$$

then the pattern predicate derived from p satisfies the typing judgment

$$X_1 :: \mathsf{Pred}\; \tau_1, \ldots, X_n :: \mathsf{Pred}\; \tau_n \vdash_T \pi(p) :: \mathsf{Pred}\; \tau$$

Definition 4.3.2.4: If p is a pattern, then $\mathit{Dom}(p) = \pi(p)\mathsf{Univ} \cdots \mathsf{Univ}$ is the pattern predicate asserting that a term matches p.
□

Notice that $\mathit{Dom}(p)$ is either a weak or an explicitly strengthened predicate depending on whether the pattern p has or does not have a $\sim$ prefix. The formula $\neg\mathit{Dom}(p)$ asserts that a term fails to match p. Thus $\neg\mathit{Dom}(p)$ is in effect, a partial definedness predicate. A term that satisfies $\neg\mathit{Dom}(p)$ must be well defined in every subterm necessary to evaluate a match with the pattern p. If p has a $(\sim)$ prefix then $\neg\mathit{Dom}(p)$, as well as $\mathit{Dom}(p)$, will be a weak predicate.

4.3.3 Patterned lambda abstraction

A patterned lambda abstraction has a pair of properties, one that it exhibits in its applications when an argument is successfully matched and another that characterizes its behavior when pattern-matching of its argument fails:

$$\frac{\Gamma,\, x_1 ::: P_1, \cdots, x_n ::: P_n \vdash t ::: Q}{\Gamma \vdash (\lambda p \texttt{->} t) \;:::\; \$(\pi(p)\, P_1 \cdots P_n \rightarrow \mathsf{Just}(Q))} \qquad \text{where } \texttt{[}x_1, \ldots, x_n\texttt{]} = \mathit{frg}\; p \tag{11}$$

$$\Gamma \vdash (\lambda p \texttt{->} t) \;:::\; \$(\neg \mathit{Dom}(p) \rightarrow \mathsf{Nothing}) \qquad \text{where } p \text{ has no } (\sim)\text{-prefix} \tag{12}$$

4.3.4 The fatbar connective

When a set of patterned applications is to be tried on an argument term, the connective ($[\!]$) with type

$$(a \rightarrow \mathsf{Maybe}\, b) \rightarrow (a \rightarrow \mathsf{Maybe}\, b) \rightarrow a \rightarrow \mathsf{Maybe}\, b$$

designates their sequential composition. Inductive rules for fatbar compositions are:

$$\frac{\Gamma \vdash \mathit{match}_1\, t ::: \mathsf{Nothing} \qquad \Gamma \vdash \mathit{matches}\, t ::: Q}{\Gamma \vdash (\mathit{match}_1 \;[\!]\; \mathit{matches})\, t ::: Q} \tag{13}$$

where Q has the kind *Maybe_Pred*.

$$\frac{\Gamma \vdash \mathit{match}_1\, t \;:::\; \mathsf{Just}(P)}{\Gamma \vdash (\mathit{match}_1 \;[\!]\; \mathit{matches})\, t \;:::\; \mathsf{Just}(P)} \tag{14}$$

4.3.5 Unguarded case expressions

The term following the keyword "of" in a case expression must be a list of matches. A simple syntactic transformation that prefixes each pattern in a case match with "λ" converts it into a patterned abstraction[1]. Erasing the "λ" inverts the transformation.

Let $\widetilde{M}$ denote the list of case matches obtained from a $[\!]$-connected list of patterned abstractions M by eliding the prefixed λ's. The following rules express the equivalence of a case expression to an application of a composite patterned abstraction. The inductive rules are:

$$\frac{\Gamma \vdash M\, t ::: \mathsf{Just}(P)}{\Gamma \vdash \mathsf{case}\; t\; \mathsf{of}\; \widetilde{M} \;:::\; P} \tag{15}$$

$$\frac{\Gamma \vdash M\, t ::: \mathsf{Nothing}}{\Gamma \vdash \mathsf{case}\; t\; \mathsf{of}\; \widetilde{M} \;:::\; \mathsf{Univ}} \tag{16}$$

4.3.6 Non-recursive let expressions

$$\frac{\Gamma,\, (\lambda \sim p \texttt{->} t_1)\, t_2 \;:::\; \mathsf{Just}(Q)}{\Gamma \vdash \mathsf{let}\; p = t_2\; \mathsf{in}\; t_1 \;:::\; Q} \qquad \text{where } \mathit{frg}(p) \cap \mathit{FV}(t_2) = \texttt{[]} \tag{17}$$

[1]The transformation produces a syntactically well-formed abstraction only when a case alternative has neither guards nor a where clause.

4.3.7 Conditional expressions

Conditional expressions provide special syntax for unguarded case expressions over a Boolean-typed discriminator.

$$\frac{\Gamma \vdash b ::: \$\mathsf{True} \qquad \Gamma \vdash t_1 ::: P}{\Gamma \vdash \mathsf{if}\ b\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2\ ::: \ P} \tag{18}$$

$$\frac{\Gamma \vdash b ::: \$\mathsf{False} \qquad \Gamma \vdash t_2 ::: Q}{\Gamma \vdash \mathsf{if}\ b\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2\ ::: \ Q} \tag{19}$$

$$\frac{\Gamma \vdash t_1 ::: P \qquad \Gamma \vdash t_2 ::: P \qquad \Gamma \vdash b ::: \$\mathsf{Univ}}{\Gamma \vdash \mathsf{if}\ b\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2\ ::: \ P} \tag{20}$$

5 Recursive definitions

A Haskell definition, $m = t_m$, justifies adding two propositions to the context of a sequent in the scope of the definition:

$$\cdots m \mathtt{===} t_m,\ (t_m ::: \$\mathsf{Univ} \Rightarrow m ::: \$\mathsf{Univ}) \vdash \cdots$$

The proposition $m \mathtt{===} t_m$ asserts the (weak) equality provided by the definition, while the implication asserts the sense of the definition.

P-logic provides a framework in which to formulate proof rules for recursively defined Haskell terms. The following rule is sound in P-logic.

$$\frac{\Gamma,\ m \mathtt{===} t_m,\ m ::: P \vdash t_m ::: \$P}{\Gamma,\ m \mathtt{===} t_m \vdash m\ ::: \ \$P} \tag{21}$$

Rule (21) encapsulates a rule for fixed-point induction in Scott logic. To discharge its antecedent clause, one must find a proof of the proposition $t_m ::: \$P$ using only the weak hypothesis $m ::: P$ and the weak equality, $m \mathtt{===} t_m$, to characterize the recursion variable.

5.1 Fixed-point properties of recursive function definitions

Typically, the properties one wishes to express of terms that arise from recursive definitions in Haskell can be characterized with least fixed-point (l.f.p.) or greatest fixed-point (g.f.p.) predicates, which are defineable in the mu-calculus. A predicate variable bound in the prefix of a fixed-point formula by either Lfp or Gfp is in scope over the entire formula[2].

A formula, H, is *admissible* for fixed-point iteration if it contains only positive occurrences of the free predicate variable that is to be bound by the fixed-point iterator. (Positive occurrence is defined in Section 7.2.6.)

5.1.1 Least fixed-point properties

To prove an l.f.p. property of a recursively defined function, the function definition may be partitioned into base cases and induction cases. For simplicity, we shall limit the number of cases to just two, which are characterized by *separation predicates* P_1 and P_2, respectively. The intended use of separation predicates is to partition the argument domain into one subset on which the function's definition yields a result without recursive invocation and a second subset on which the definition must invoke recursion to return a result.

[2]In P-logic, we use the identifiers Lfp and Gfp as substitutes for the greek letters μ and ν used in mathematical treatments of the mu-calculus.

A rule with which an l.f.p. property can be proven is

$$\frac{\begin{array}{l}\Gamma,\, m ::: \mathsf{Univ} \;\vdash t_m ::: \$(P_1 \rightarrow H) \\ \Gamma,\, m ::: \$(P_1 \,\vee\, P_2 \rightarrow X) \vdash t_m ::: \$(P_2 \rightarrow H)\end{array}}{\Gamma,\, m \mathop{===} t_m \vdash m ::: \$(P_1 \,\vee\, P_2 \rightarrow \mathsf{Lfp}\, X \bullet H)} \tag{22}$$

where X is a predicate variable that may occur in H but X does not occur in P_1 or P_2 and $X \notin FV(\Gamma)$.

The first antecedent clause provides a base case for induction. The assertion $m ::: \mathsf{Univ}$ in the context of the first antecedent ensures that its conclusion cannot depend upon any specific property assumed of the term variable m. This antecedent clause asserts that whenever term t_m is applied to an argument restricted by property P_1, the application can be proved to satisfy the predicate formula H without assuming a property of the function to hold at a recursive application.

The second antecedent clause provides an induction step. The inductive assumption asserts the property H of an application of t_m to an argument restricted by the property P_2. The term variable m may occur in t_m and the predicate variable X may occur in H.

The consequent of this rule asserts an l.f.p. property of a recursively defined function m when its definition, $m \mathop{===} t_m$, is added to the context. Rule (22) is a means by which to discharge the antecedent clause of rule (21) for a recursive function definition.

5.1.2 Greatest fixed-point properties

To prove a g.f.p. property stated with term congruences, we make use of the fact that Haskell data constructors are uniquely invertible. Thus a constructor pattern provides, implicitly, deconstruction functions that project out the component subterms of a constructed term matching the pattern. The semantics of pattern matching in Haskell relies upon this isomorphism between a constructed term and its components. So do the term congruences of P-logic.

To prove a g.f.p. property, the following rule is sound. In this rule, we have again assumed that the recursive and non-recursive instances of a function definition are separated by a partition of its argument domain.

$$\frac{\begin{array}{l}\Gamma,\, m ::: \mathsf{Univ} \;\vdash t_m ::: \$(P_1 \rightarrow H) \\ \Gamma,\, t_m ::: \$(P_2 \rightarrow H) \vdash m ::: \$(P_1 \,\vee\, P_2 \rightarrow X)\end{array}}{\Gamma,\, m \mathop{===} t_m \vdash m ::: \$(P_1 \,\vee\, P_2 \rightarrow \mathsf{Gfp}\, X \bullet H)} \tag{23}$$

where the predicate variable X may occur in H but does not occur in P_1 or P_2 and $X \notin FV(\Gamma)$.

The first antecedent is the same as that for the l.f.p. rule and accounts for applications that do not invoke recursive calls of the function. The second antecedent asserts that when an application of t_m to an argument with property P_2 satisfies H, the term variable m satisfies property X.

Since the predicate in the conclusion of the consequent is defined by a g.f.p. formula, the so-called "base case" represented by the domain satisfying P_1 may be empty. For an l.f.p. formula, the base case could not be empty or else the formula would be (weakly) satisfied only by the bottom element of the computational domain.

6 Monads

The monad class in Haskell specifies an operator signature containing: $(\mathsf{>>=}) :: \mathsf{M}(a) \rightarrow (a \rightarrow \mathsf{M}(b)) \rightarrow \mathsf{M}(b)$ and $\mathsf{return} :: a \rightarrow \mathsf{M}(a)$. Reasoning about programs whose results are typed in the monad class is based upon an assumption that in every instance of the class, these operators satisfy the equations

(*Left-id*)	$\forall t :: \mathsf{M}(a).\, (t \mathsf{>>=}\ \mathsf{return}) \mathop{===} t$
(*Right-id*)	$\forall x :: a.\ (\mathsf{return}\ x \mathsf{>>=} f) \mathop{===} f$
(*Assoc*)	$\forall t :: \mathsf{M}(a).\, (t \mathsf{>>=} \lambda x \mathsf{->} (g\ x \mathsf{>>=} f)) \mathop{===} ((t \mathsf{>>=} g) \mathsf{>>=} f)$

Let's call the conjunction of this set of properties $Monad\text{-}eqs_{\mathsf{M}}$.

When reasoning about expressions typed in a monad, we must account for properties of the the operational context that affects operations in the monad, as well as properties associated with "pure" expressions. *P*-logic accounts for that state of the operational context with predicate formulas conjoined with the normal predicate formulas in a type $\mathsf{Pred}(\mathsf{M}\,a)$. Formulas in this type are pairs, (P, W_{M}), where $P :: \mathsf{Pred}(a)$ and $W_{\mathsf{M}} :: \mathsf{Pred}_{\mathsf{M}}$, where $\mathsf{Pred}_{\mathsf{M}}$ is a distinct predicate type for each monad instance, M.

We offer no comprehensive explanation of what it means for an expression typed in a monad to satisfy a monadic predicate. Meanings can be given by stating rules for the operators of particular monads.

The context of a judgment may include a component which asserts properties of the operational context of a computation. We shall designate such a component proposition by subscripting it with M in the rules that follow.

6.1 Rules for monadic computations

The following rules express properties of computations formed with the standard monadic operators:

$$(\textit{Monad-unit}) \qquad \frac{\Gamma \vdash \$Monad\text{-}eqs_{\mathsf{M}} \qquad \Gamma \vdash t ::: P}{\Gamma,\, W_{\mathsf{M}} \vdash \mathsf{return}\; t ::: (P, W_{\mathsf{M}})} \tag{24}$$

$$(\textit{Monad-bind}) \qquad \frac{\Gamma \vdash \$Monad\text{-}eqs_{\mathsf{M}} \quad \Gamma,\, S_{\mathsf{M}} \vdash t_1 ::: (P, V_{\mathsf{M}}) \quad \Gamma,\, V_{\mathsf{M}} \vdash t_2 ::: P \to (Q, W_{\mathsf{M}})}{\Gamma,\, S_{\mathsf{M}} \vdash t_1 \mathbin{\mathsf{>>=}} t_2 ::: (Q, W_{\mathsf{M}})} \tag{25}$$

7 Foundations

In this section we specify a meaning for *P*-logic. An interpretation, Φ, maps propositional formulas in the logic into propositions that refer to a specific model in which the truth of each proposition can be judged. An interpretation has two parts, (a) a model for the term language and (b) a model for predicate formulas of *P*-logic.

A Haskell expression stands for its denotation in a poset domain. As Haskell is strongly typed, the meaning of an expression with a type τ can be given in terms of a domain specific to its type. Accordingly, we shall assume a semantics function that gives meanings to type derivations, in which every term is paired with a type.

Let $\mathcal{E}[\![\,\_\,]\!] :: (\mathit{Term} \times \mathit{Type}) \to \mathit{Env} \to \mathcal{D}$ be a meaning function that maps every well-formed Haskell expression of type τ to its denotation in a domain that models its type, where $\mathit{Env} = \mathit{Var} \times \mathit{Type} \to \mathcal{D}$.

7.1 A semantics interpretation of P-logic

To say exactly what formulas in *P*-logic mean, formulas will be interpreted as characteristic predicates of sets (posets) in an abstract domain for Haskell's semantics. Each formula is interpreted in a Haskell type (or type scheme, as we shall admit type variables). We use the following notation to distinguish formulas and terms from their meanings:

$\lceil\tau\rceil_\perp$	is the "lifted" set of interpretations of terms of type τ
$\lceil\tau\rceil$	is the set of interpretations of terms of type τ, excluding $\perp_\tau$ (i.e., $\lceil\tau\rceil = \lceil\tau\rceil_\perp \setminus \{\perp_\tau\}$)
$c_{\lceil\tau\rceil}$	is the interpretation of the constant symbol c in the type τ
$[\![P]\!]^\tau_\perp$	$\subseteq \lceil\tau\rceil_\perp$
$[\![P]\!]^\tau$	$\subseteq \lceil\tau\rceil$

7.1.1 Universal predicates

The predicate constants Univ and UnDef represent the universal predicate and the unsatisfiable predicate in each type. The interpretations of these predicates are:

$$[\![\mathsf{Univ}]\!]^{\tau}_{\perp} = \lceil\tau\rceil_{\perp} \qquad [\![\mathsf{UnDef}]\!]^{\tau}_{\perp} = \{\perp_{\tau}\}$$

7.1.2 Constant sections as predicates

A predicate that asserts equality with a constant is satisfied by the denotation of the specified constant.

$$[\![(\texttt{==}\ c)]\!]^{\tau}_{\perp} = \{\perp_{\tau},\, c_{\lceil\tau\rceil}\}$$

Inequality predicates are defined for numeric types in the *Real* class.

$$\mathit{Real}\,\tau \texttt{ => } [\![(<c)]\!]^{\tau}_{\perp} = \{x \in \lceil\tau\rceil_{\perp} \mid x <_{\lceil\tau\rceil} c_{\lceil\tau\rceil}\}$$

where $<_{\lceil\tau\rceil}$ is the order inequality in the algebra of $\lceil\tau\rceil$.

7.1.3 Term congruence predicates

Term congruence predicates are formed with the constructors of datatypes, lifted to act as predicates. The declaration of a datatype, τ, (or a datatype constructor) produces a finite set of signatures, Σ^{τ}_{k}, each of which which contains the arity-k constructors of the datatype, each paired with the k-tuple of its argument types.

$$((C,\,(\tau_1,\ldots,\tau_k)) \in \Sigma^{\tau}_{k}) \;\Rightarrow\; [\![C\,P_1\cdots P_k]\!]^{\tau}_{\perp} = \{C_{\lceil\tau\rceil}\,t_1\cdots t_k \mid t_1 \in [\![P_1]\!]^{\tau_1}_{\perp} \wedge \ldots \wedge t_k \in [\![P_k]\!]^{\tau_k}_{\perp}\} \cup \{\perp_{\tau}\}$$

Note that the above interpretation is given for non-strict datatype constructors.

7.1.4 Arrow predicates

An arrow predicate characterizes a property of a function-typed term. We can read a proposition such as $M ::: P \to Q$ as the assertion "when M is applied to an argument that has property P, the application has property Q". We call the subformula to the left of the arrow the domain predicate and that to its right the range predicate.

$$[\![P \to Q]\!]^{\tau_1\to\tau_2} = \{f \in \lceil\tau_1\rceil_{\perp} \to \lceil\tau_2\rceil_{\perp} \mid \forall x \in [\![P]\!]^{\tau_1}_{\perp} \bullet f\,x \in [\![Q]\!]^{\tau_2}\}$$

Notice that for a function which satisfies (strongly) an arrow predicate, it must yield a well-defined result at every argument value covered by the domain predicate.

The argument to a non-strict function is only required to satisfy a weak interpretation of the domain predicate. To characterize strict functions, the domain predicate must be explicitly strengthened.

7.1.5 Maybe predicates

Predicate types in the *Maybe_Pred* kind distinguish the codomain types of *match* clauses from those of function-typed expressions. They are modeled by tagged semantic values.

$$[\![\mathsf{Just}\ P]\!]^{\mathrm{Maybe}\,\tau} = \{(1,v) \mid v \in [\![P]\!]^{\tau}_{\perp}\}$$
$$[\![\mathsf{Nothing}]\!]^{\mathrm{Maybe}\,\tau} = \{(0,\perp_{\tau})\}$$

7.1.6 Predicate conjunction and disjunction

$$\llbracket P_1 \wedge P_2 \rrbracket^{\tau}_{\perp} = \llbracket P_1 \rrbracket^{\tau}_{\perp} \bigcap \llbracket P_2 \rrbracket^{\tau}_{\perp}$$
$$\llbracket P_1 \vee P_2 \rrbracket^{\tau}_{\perp} = \llbracket P_1 \rrbracket^{\tau}_{\perp} \bigcup \llbracket P_2 \rrbracket^{\tau}_{\perp}$$

In the strong interpretations of conjunction and disjunction, exclusion of the bottom value distributes into the subformulas. That is,

$$\llbracket P_1 \wedge P_2 \rrbracket^{\tau} = \llbracket P_1 \rrbracket^{\tau} \bigcap \llbracket P_2 \rrbracket^{\tau}$$
$$\llbracket P_1 \vee P_2 \rrbracket^{\tau} = \llbracket P_1 \rrbracket^{\tau} \bigcup \llbracket P_2 \rrbracket^{\tau}$$

7.1.7 Equality

The binary equality predicate is defined in terms of equality in the underlying semantics domain for each type. A strong interpretation of equality requires that two terms denote equal, non-bottom elements of the domain.

$$\llbracket (\texttt{===}) \rrbracket^{\tau}_{\perp} = \{u, v \mid u \in \lceil \tau \rceil_{\perp} \wedge v \in \lceil \tau \rceil_{\perp} \wedge u = v\}$$

7.1.8 Fixed-point formulas

The interpretations of least (greatest) fixed-point formulas are given in terms of infinite unions (intersections) of interpretations of finitely iterated syntactic substitutions of the matrix in place of the recursion variable, X. The zeroth iteration is specified by definition to be UnDef in the interpretation of an l.f.p. formula, and Univ in the interpretation of a g.f.p. formula. Thus the interpretations are dual.

$$\llbracket \mathsf{Lfp} X.\, H \rrbracket^{\tau}_{\perp} = \bigcup_{j=0}^{\infty} \llbracket H^j \rrbracket^{\tau}_{\perp} \quad \text{where} \quad \begin{array}{l} H^0 = \mathsf{UnDef} \\ H^{j+1} = H[H^j/X] \end{array}$$

$$\llbracket \mathsf{Gfp} X.\, H \rrbracket^{\tau}_{\perp} = \bigcap_{j=0}^{\infty} \llbracket H^j \rrbracket^{\tau}_{\perp} \quad \text{where} \quad \begin{array}{l} H^0 = \mathsf{Univ} \\ H^{j+1} = H[H^j/X] \end{array}$$

7.1.9 Example: Tail-strict lists

Consider the l.f.p. formula $\mathsf{Lfp} X.\, \texttt{[]} \vee (A : \$X)$ where A is the unique, nullary constructor of the datatype declared by:

data $unitA = A$

The sequence of iterated formulas for the l.f.p. interpretation begins as:

$\mathsf{UnDef},\ (\texttt{[]} \vee (A : \$\mathsf{UnDef})),\ (\texttt{[]} \vee (A : \$(\texttt{[]} \vee (A : \$\mathsf{UnDef})))), \ldots$

in which every predicate substituted for the predicate variable in an iteration is strengthened because the context of the predicate variable was strengthened at its definition.

Consider, for example, the interpretation of the second formula in the sequence above:

$$\begin{aligned}
&[\![(\texttt{[]} \vee (A : \$\mathsf{UnDef}))]\!]_\perp \\
&\quad = [\![\texttt{[]}]\!]_\perp \cup [\![(A : \$\mathsf{UnDef})]\!]_\perp \\
&\quad = \{\texttt{[]}\} \cup [\![(A : \$\mathsf{UnDef})]\!]_\perp \\
&\quad = \{\texttt{[]}\} \cup \{\perp\} \cup \{(a : as) \mid a \in [\![A]\!]_\perp \text{ and } as \in [\![\$\mathsf{UnDef}]\!]_\perp\} \\
&\quad = \{\texttt{[]}\} \cup \{\perp\} \cup \{\,\} \\
&\quad = \{\perp, \texttt{[]}\}
\end{aligned}$$

The sequence starts with the singleton set containing only $\perp$ then adds the set containing $\perp$ and `[]`, then the set containing $\perp$, `[]`, and all lists of unit length, and so on. However, it never contains an element in which the tail of a list is the bottom element, in consequence of the strengthened predicate, $\$\mathsf{UnDef}$, in the tail position of the constructor congruence.

$$\{\perp\},\ \{\perp, \texttt{[]}\},\ \{\perp, \texttt{[]}, \texttt{[}\perp\texttt{]}, \texttt{[}A\texttt{]}\},\ \{\perp, \texttt{[]}, \texttt{[}\perp\texttt{]}, \texttt{[}A\texttt{]}, \texttt{[}\perp,\perp\texttt{]}, \texttt{[}\perp, A\texttt{]}, \texttt{[}A, \perp\texttt{]}, \texttt{[}A, A\texttt{]}\}, \ldots$$

The limit of this sequence is the set of all finite *unitA* lists.

If we omit tail-strictness from the specification, the formula $\mathsf{Lfp}X.\ \texttt{[]} \vee (A : X)$, unfolded into a sequence of iterated substitutions begins as:

$$\mathsf{UnDef},\ (\texttt{[]} \vee (A : \mathsf{UnDef})),\ (\texttt{[]} \vee (A : (\texttt{[]} \vee (A : \mathsf{UnDef})))), \ldots$$

and has the interpretation

$$\begin{aligned}
&\{\perp\},\ \{\perp, \texttt{[]}, (\perp : \perp), (A : \perp)\}, \\
&\qquad \{\perp, \texttt{[]}, \texttt{[}\perp\texttt{]}, \texttt{[}A\texttt{]}, (\perp : (\perp : \perp)), (A : (\perp : \perp)), (\perp : (A : \perp)), (A : (A : \perp))\}, \ldots
\end{aligned}$$

This sequence of interpretations is much richer than that for the tail-strict formula, as the component sets include least approximations of extensions of the lists. In this sequence, unlike the previous one, every element of the $i + 1^{st}$ set ($i > 1$) is approximated in the domain ordering by some non-bottom element of the i^{th} set, and approximates some element in the $i + 2^{nd}$ set. The limit of the sequence is a set containing all of the tail-strict, finite lists of *unitA* elements but it also contains all of their partially-evaluated approximations, including approximations of infinite lists! Since the approximating elements do not terminate in the nil list, all infinite lists of type [*unitA*] will appear in the limit.

7.2 What it means to say that a term satisfies a predicate

An atomic, unary predicate in P-logic is either one of the predicate constants Univ and UnDef or is formed of a constant section expression or a nullary data constructor, implicitly lifted to the status of a predicate. Compound predicates are composed with logical connectives, the arrow predicate constructor, k-ary data constructors, and least and greatest fixed-point bindings.

We shall say how predicates of each form are satisfied with respect to a context. The properties assumed in the context of a sequent clause constrain the possible environments that give meanings to term variables. We shall define a refinement relation between an environment, ρ, and a list of conjoined assertions, Γ.

First, however, we must specify how an assertion is interpreted with respect to an environment that assigns meanings to term variables.

Definition 7.2.1:

$$\begin{array}{l}
\mathcal{F} :: \mathsf{Prop} \to \mathit{Env} \to \{\mathit{true}, \mathit{false}\} \\
\mathcal{F}\,\|t ::: P :: \mathsf{Pred}\,\tau\|\,\rho \equiv \mathcal{E}\,\|t,\tau\|\,\rho \in \|P\|^{\tau}_{\perp} \\
\mathcal{F}\,\|t_1 \texttt{===}_{\tau}\, t_2\|\,\rho \equiv \mathcal{E}\,\|t_1,\tau\|\,\rho = \mathcal{E}\,\|t_2,\tau\|\,\rho \\
\mathcal{F}\,\|P^{(k)}\,t_1 \cdots t_k\|\,\rho \equiv (\mathcal{E}\,\|t_1,\tau_1\|\,\rho, \ldots, \mathcal{E}\,\|t_k,\tau_k\|\,\rho) \in \|P^{(k)}\|^{\tau_1,\cdots,\tau_k}_{\perp} \\
\qquad \text{where } P^{(k)} :: \mathsf{Pred}\,\tau_1 \cdots \tau_k
\end{array}$$

Definition 7.2.2: We say that an environment, ρ, *refines* a context, Γ, (written as $\rho \lhd \Gamma$) if

$$\bigwedge(\mathsf{map}\,(\mathcal{F}\,\|\,\_\,\|\,\rho)\,\Gamma) \equiv \mathit{true}$$

where Γ is a list of assertions and $\bigwedge$ is the list homomorphism of conjunction, $(\wedge)$.

An environment, ρ, *supports* a conclusion, Ξ, (written as $\rho \rhd \Gamma$) if

$$\bigvee(\mathsf{map}\,(\mathcal{F}\,\|\,\_\,\|\,\rho)\,\Gamma) \equiv \mathit{true}$$

In this definition, we have abused notation by using the Haskell function map to designate a functor from *Set* to a category of finitely iterated products over *Set*.

Definition 7.2.3: A term t *weakly satisfies* a predicate, $P :: \mathsf{Pred}\,\tau$, in a context represented by Γ, if
$\forall \rho.\,(\rho \lhd \Gamma) \Rightarrow \mathcal{E}\|t,\tau\|\,\rho \in \|P\|^{\tau}_{\perp}$.

A term t *strongly satisfies* a predicate, $\$P :: \mathsf{Pred}\,\tau$, in a context represented by Γ, if
$\forall \rho.\,(\rho \lhd \Gamma) \Rightarrow \mathcal{E}\|t,\tau\|\,\rho \in \|P\|^{\tau}$.

□

7.2.1 Equality

Two terms, t_1 and t_2, satisfy the equality predicate $(\texttt{===}) :: \mathsf{Pred}\ \tau\ \tau$ with respect to a context, Γ, if $\forall \rho.\,(\rho \lhd \Gamma) \Rightarrow (\mathcal{F}\,\|t_1 \texttt{===}_{\tau}\, t_2\|\,\rho = \mathit{true})$. The strengthened equality, $t_1 \,\$\texttt{===}_{\tau}\, t_2$ is satisfied if both (i) $t_1 \texttt{===}_{\tau}\, t_2$ with respect to Γ and (ii) $\forall \rho.\,(\rho \lhd \Gamma) \Rightarrow \mathcal{E}\|t_1,\tau\|\,\rho \neq \perp_{\tau}$.

7.2.2 Type-indexed orderings

A total ordering $(<)$ is defined on instances of the *Ord* class. The inequality operator is lifted to a binary predicate in *P*-logic. Two terms, t_1 and t_2, belonging to a base type τ in the *Ord* class satisfy the predicate $(<) :: \mathsf{Pred}\ \tau\ \tau$ in the context Γ iff $\forall \rho.\,(\rho \lhd \Gamma) \Rightarrow \mathcal{E}\|t_1,\tau\|\,\rho <_{\tau} \mathcal{E}\|t_2,\tau\|\,\rho$, where $<_{\tau}$ is the inequality operator in the domain $\lceil \tau \rceil$. An inequality predicate is weakly satisfied if either it is strongly satisfied or both of the terms being compared denote $\perp$.

7.2.3 Universal predicates

The predicate $\mathsf{Univ} :: \mathsf{Pred}\ \tau$ is satisfied by every term of type τ. It is strongly satisfied by a term, t, with respect to a context, Γ, if $\forall \rho.\,(\rho \lhd \Gamma) \Rightarrow \mathcal{E}\|t,\tau\|\,\rho \neq \perp_{\tau}$.
The predicate $\mathsf{UnDef} :: \mathsf{Pred}\ \tau$ is satisfied by a term $t :: \tau$ with respect to a context, Γ, if $\forall \rho.\,(\rho \lhd \Gamma) \Rightarrow \mathcal{E}\|t,\tau\|\,\rho = \perp_{\tau}$. No term strongly satisfies UnDef.

7.2.4 Nullary constructors as predicates

The satisfaction criteria for a nullary constructor as a predicate is:

$$\Gamma \models t ::: C^{(0)} :: \mathsf{Pred}\ \tau \equiv \forall \rho.\, (\rho \lhd \Gamma) \Rightarrow \mathcal{E}[\![t, \tau]\!]\, \rho = C^{(0)}_{\lceil \tau \rceil} \text{ or } \mathcal{E}[\![t, \tau]\!]\, \rho = \bot_\tau$$

Strong satisfaction criteria are the same except that they disallow the alternative that a term might denote $\bot_\tau$ in some environment that refines Γ.

7.2.5 Constructor congruence predicates

A constructor congruence predicate is an application of a lifted k-ary constructor to k predicate formulas. The satisfaction criterion for a congruence predicate formed with a non-strict constructor is:

$$\begin{aligned}\Gamma \models t ::: C^{(k)}\, P_1 \cdots P_k :: \mathsf{Pred}\ \tau \equiv \forall \rho.\, (\rho \lhd \Gamma) \Rightarrow \exists v_1, \ldots, v_k.\, (&\mathcal{E}[\![t, \tau]\!]\, \rho = C^{(k)}_{\lceil \tau \rceil}\, v_1 \cdots v_k \\ &\text{and } v_i \in [\![P_i]\!]_\bot \qquad (1 \le i \le k) \\ \text{or } &\mathcal{E}[\![t, \tau]\!]\, \rho = \bot_\tau\end{aligned}$$

Strong satisfaction is similar but imposes the additional condition that $\forall \rho.\, (\rho \lhd \Gamma) \Rightarrow \mathcal{E}[\![t, \tau]\!]\, \rho \neq \bot_\tau$.

Weak satisfaction of a congruence predicate built with a strict constructor, $C^{(k)}!$, is defined by:

$$\begin{aligned}\Gamma \models t ::: C^{(k)}!\, P_1 \cdots P_k :: \mathsf{Pred}\ \tau \equiv \forall \rho.\, (\rho \lhd \Gamma) \Rightarrow \exists v_1, \ldots, v_k.\, (&\mathcal{E}[\![t, \tau]\!]\, \rho = C^{(k)}_{\lceil \tau \rceil}\, v_1 \cdots v_k \\ &\text{and } v_i \in [\![P_i]\!] \qquad (1 \le i \le k) \\ \text{or } &\mathcal{E}[\![t, \tau]\!]\, \rho = \bot_\tau\end{aligned}$$

in which the arguments of a constructed term must satisfy the strong interpretations of the respective predicates.

7.2.6 Recursive predicates

A *predicate context* is a predicate formula in which can be embedded one or more occurrences of an arbitrary predicate[3], X. A predicate context can have *parity*, positive or negative according to whether an embedded predicate variable is nested under an even or an odd number of negation symbols, counting zero as an even number.

An inductive definition of linear predicate contexts is:

Definition 7.2.6.1 Positive and negative linear predicate contexts (respectively) are specified by:

$$\begin{aligned}H[\,] &::= [\,] \mid \neg N[\,] \mid P \vee H[\,] \mid P \wedge H[\,] \mid P \rightarrow H[\,] \mid C^n\, P_1 \ldots P_{k-1}\, H[\,]\, P_{k+1} \ldots P_n \quad (1 \le k \le n) \\ N[\,] &::= \neg H[\,] \mid P \vee N[\,] \mid P \wedge N[\,] \mid P \rightarrow N[\,] \mid C^n\, P_1 \ldots P_{k-1}\, N[\,]\, P_{k+1} \ldots P_n \quad (1 \le k \le n)\end{aligned}$$

where $C^{(n)}$ is an n-ary constructor symbol. Symmetry of the connectives $(\wedge)$ and $(\vee)$ is assumed.
□

A least fixed-point predicate formula is the limit of a disjunctive sequence of predicate contexts for UnDef, i.e. $H[\mathsf{UnDef}] \vee H[H[\mathsf{UnDef}]] \vee H[H[H[\mathsf{UnDef}]]] \vee \cdots$, where $H[\,]$ is a positive linear predicate context. This intuition leads us to a formal definition of satisfaction of an l.f.p. formula:

$$\Gamma \models t ::: \mathsf{Lfp}X.\, H[X] \equiv \exists n \in \mathit{Nat}.\, \Gamma \models t ::: H^n[\mathsf{UnDef}]$$

Dually, satisfaction of a g.f.p. formula is defined by:

$$\Gamma \models t ::: \mathsf{Gfp}X.\, H[X] \equiv \forall n \in \mathit{Nat}.\, \Gamma \models t ::: H^n[\mathsf{Univ}]$$

[3]In this definition, type conformity is implicit. Predicate typing annotations have been omitted for brevity.

7.3 Soundness

A sequent $\Gamma \vdash \Xi$ is *valid* for an interpretation Φ iff $(\models_\Phi \Gamma) \;\Rightarrow\; (\models_\Phi \Xi)$.

For a standard interpretation as outlined in Section 7.1, indexed by an environment, ρ, the validity of a sequent $\Gamma \vdash \Xi$ is expressed by the logical implication

$$\forall \rho.\, \rho \lhd \Gamma \Rightarrow \rho \rhd \Xi$$

A sequent rule, $\frac{antecedents}{consequent}$, is *sound* with respect to an interpretation Φ if in every context for which each of its *antecedents* is valid for Φ, its *consequent* is also valid for Φ.

Theorem: P-logic is sound with respect to the standard interpretation outlined in Section 7.1.
Proof: Each of the logical axioms is valid and each of the rules is sound. (See Appendix I for details.)

8 Experiments with P-logic

We have conducted several experiments in which P-logic has been used to state properties expected of small Haskell functions. Proofs of the properties were constructed by hand, using rules of P-logic augmented by *ad hoc* rules to express a few properties needed of integer arithmetic and the ordering of integer values. The proofs were then checked by coding the proof rules and the strategies employed in these proofs in *Stratego* [VeAB98, Vis99, Vis01b, Vis01a]. Two interesting examples are briefly described below.

Example 1: ***length*** **of a catenated list**

Defining equations for the Haskell functions *length* and `(++)` are:

```
length [] = 0
length (x:xs) = 1 + length xs

[] ++ ys = ys
(x:xs) ++ ys = x:(xs ++ ys)
```

The proof context (assumptions), Γ_{lngth}, contained the following clauses

$(\Gamma_{\text{lngth}}.0)$ $\quad ys ::: \mathit{Finite\text{-}list}$

$(\Gamma_{\text{lngth}}.1)$ $\quad \mathit{length}\ \texttt{[]} =_{\mathit{def}}\ 0$
$(\Gamma_{\text{lngth}}.2)$ $\quad \mathit{length}\ (x : xs) =_{\mathit{def}}\ 1 + \mathit{length}\ xs$

$(\Gamma_{\text{lngth}}.3)$ $\quad \texttt{[]} \mathbin{+\!+} ys =_{\mathit{def}}\ ys$
$(\Gamma_{\text{lngth}}.4)$ $\quad (x : xs) \mathbin{+\!+} ys =_{\mathit{def}}\ bx : (xs \mathbin{+\!+} ys)$

where the predicate *Finite-list* is defined in Sec. (2.1).

We also assumed monoid rules for ((+),0), as an *ad hoc* assumption about the algebra of integer arithmetic.

The assertion to be proved is

$$\Gamma_{\text{lngth}},\ xs ::: \mathit{Finite\text{-}list},\ ys ::: \mathit{Finite\text{-}list} \vdash \mathit{length}\ (xs \mathbin{+\!+} ys) \mathbin{===} \mathit{length}\ xs\ +\ \mathit{length}\ ys$$

In the proof of this assertion we employed an specific induction rule for equality on terms of list datatypes. (This specialized rule was not shown in the preceding sections.) This rule is:

$$\frac{\begin{array}{l}\Gamma \vdash M\,[\texttt{[]}] \mathbin{===} N\,[\texttt{[]}] \\ \Gamma,\ x ::: \mathsf{Univ},\ xs ::: \$\mathit{Finite\text{-}list} \vdash (M\,[xs] \mathbin{===} N\,[xs]) \Rightarrow M\,[(x : xs)] \mathbin{===} N\,[(x : xs)]\end{array}}{\Gamma,\ xs ::: \$\mathit{Finite\text{-}list} \vdash_{WS} M\,[xs] \mathbin{===} N\,[xs]}$$

Upon substituting the context expression ($\mathit{length}\ \_ + \mathit{length}\ \mathit{ys}$) for $M[\_]$ and the context expression $\mathit{length}\ (\_ \mathbin{+\!+} \mathit{ys})$ for $N[\_]$, the antecedents of the inductive list-equality rule can be proved from the properties assumed in Γ_{lngth}.

Encoded in Stratego, the support for this proof comprises some 26 strategy declarations in addition to rules of the logic. These strategies largely manipulate terms and definitions in the Haskell abstract syntax but also apply rules of the logic to terms that are in the form expected by the rules. The term manipulations implemented by the strategies include tupling and untupling of multiple arguments to a function, unfolding and refolding definitions, and implementing substitutions for free variables in a term.

Example 2: A property of insertion in a list of integer values.

A Haskell function for non-duplicating, ordered insertion of an integer in a list can be defined recursively as follows:

```
insert :: Int -> [Int] -> [Int]
insert a [] = [a]
insert a ys@(y:_) | a < y  = a:ys
                  | a == y = ys
insert a (y:ys) = y:insert a ys
```

A sequent that one might wish to prove about *insert* is

$$\mathsf{All}\ n :: \mathit{Int}.\ \mathsf{All}\ \mathit{xs} :: [\mathit{Int}].\ \mathit{xs} ::: \mathit{All\_elts}\ \$\mathsf{Univ} \Rightarrow n\ \text{`}\mathit{IsInserted}\text{`}\ (\mathit{insert}\ n\ \mathit{xs})$$

where the unary predicate *All_elts* is defined in Section 2.1 and the binary predicate *IsInserted* is defined in Section 3.5. The list into which an insertion is made is not restricted to be finite, however. Intuitively, the insertion property should hold even for insertions into an infinite list of integers.

Indeed, this assertion can be proven in *P*-logic, although the proof is somewhat longer than the proof of the assertion in Example 1, using coinductive proof rules to establish a g.f.p. property. A proof of the insertion property has been duplicated with Stratego, directed by appropriate programmed strategies and using the rules of *P*-logic.

8.1 Related work

Simon Thompson's early effort to give a verification logic [Tho89, Tho95] for Miranda, a lazy, functional language that was a predecessor to Haskell, exposed many of the difficulties inherent in adapting a first-order predicate calculus for use as a verification logic. The logic for Miranda employs quantification operators that bind variables to range only over *defined* terms, or over *finite* structures of a datatype. The meanings of such quantifiers are extra-logical; they cannot be defined in the logic itself.

Sparkle [dvP01] is a verification tool for *Clean* [Bvv+87, Pla00], 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.

Temporal logics [Pnu77, Lam80] for programs were developed to provide a formalism in which to express properties of programmed processes. Linear temporal logic (LTL) supports property assertions quantified over the temporal traces of a process, where by a trace, we mean a sequence of states entered in a process trajectory as the progression of time is measured in discrete steps. In a branching temporal logic, such as CTL [CE81, EC82], properties can be quantified over possible forward state trajectories of a process, specifically allowing for choice points at which alternate continuations of a trajectory are possible.

It is of interest to observe, however, that nothing ties the formal definitions of LTL and CTL to a notion of time, as we perceive it in the physical universe. Rather, LTL is a modal logic in which modal predicates are interpreted over sets of semi-unbounded sequences, rather than over unstructured sets of elements. CTL is a modal logic in which predicates are interpreted over finitely-branching, unbounded, node-labeled trees. Such logics can be applied whenever a universe of sequences or of trees furnishes an appropriate model.

In the modal mu-calculus [Koz83], modal operators such as those of LTL and CTL can be defined recursively, using least and greatest fixed-point operators to express quantifications with respect to depth in a sequence or a term, in conjunction with the modal operators that quantify over the possible paths through a term.

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

A different kind of modality is used in P-logic to characterize normalization of terms by differentiating strong and weak satisfaction criteria. The introduction of this modality was inspired by a three-valued propositional logic, *WS*-logic [Owe97], which conservatively extends classical propositional logic, with the notable exception that the trivial sequent, $P \vdash P$ is not sound[4].

A modality analogous to the *weak—strong* modality of P-logic was introduced by Larsen [Lar89] 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 [HJS01] generalized Larsen's analysis and provided a semantic interpretation of the modality in a more general framework. Their semantic interpretation of a predicate is a pair of power-domain elements, $(P_\perp, P_\top)$, where $P_\perp$ is downward-closed and $P_\top$ is upward-dense. These interpretations are used in modeling *may* and *must* properties, respectively. This general characterization of predicate interpretations also applies to the weak and strong notions of predicate satisfaction that we have used in P-logic.

9 Conclusions and future work

This paper is the initial presentation of P-logic, a comprehensive, integrated verification logic for the programming language Haskell 98. P-logic is intended to be tightly integrated with Haskell—so much so, that assertions made in the logic can be embedded into a program text at roughly the same points that declarations or explicit typing assertions are allowed. The terms whose properties can be asserted in the logic are exactly the Haskell expressions whose variables are in scope at the point that an assertion is embedded. Formulas can be unary, binary, or in general, k-ary. Equality is a distinguished binary predicate form.

The formula language of P-logic is a modal mu-calculus whose modalities are the modalities of Haskell terms. In the modality of partial definedness, a property can hold of a term even if it denotes $\perp$, the undefined value in its semantics domain. In the modality of strong definedness, a property holds only of well-defined values. These modalities are appropriate for properties asserted in non-strict and strict contexts, respectively.

[4]Notice that under the "swap" rule of classical sequent logic, $P \vdash P \;\equiv\; \vdash \neg P, P$. However, the disjunctive propositions in the conclusion of the consequent fail to saturate the possible valuations of P in three-valued *WS* logic.

The data constructors that can be declared in a Haskell program introduce additional modalities. Data constructors are "lifted" to predicate constructors in P-logic through a congruence implicit in the isomorphism between a constructed term and the tuple of subterms from which it is constructed. A lifted data constructor applied to a tuple of unary predicate formulas propagates an assertion of these formulas to the subterms of a data value constructed with the same constructor.

Together with the least and greatest fixed-point binding operators of the mu-calculus, the rich set of predicate constructors affords P-logic the ability to define modalities for specific data structures in Haskell. P-logic supports the definition of type-specific modal operators for each Haskell datatype.

P-logic is designed to support reasoning about observable properties of programs in preference to semantic equivalence. Thus there is no direct way to assert that "function f is everywhere equal to function g", when f and g may be partial functions. Instead, one must assert "f is equivalent to g *and both are defined* everywhere on the common domain characterized by P". The latter property is observable in principle (if the domain is enumerable), whereas the former is not.

Type-indexed predicates in P-logic provide a means to reason about properties of expressions that contain overloaded operators. A logic of type-indexed predicates is coherent with a semantics that gives meanings to type derivations—fully type-annotated expressions. Since the logic makes no distinction between compile-time and run-time, it isn't necessary to specify a mechanism for dynamic type resolution of overloaded expressions. Nor is it necessary to do so in a semantics for Haskell, other than to specify that such a mechanism is required in a conforming operational semantics. The proof rules (22 and 23) that establish fixed-point properties of a function rely upon separation predicates to characterize the domains on which an application of the function can invoke recursion and on which it must not. The separation predicate characterizing the domain of recursion must be guessed or supplied by an oracle to support a proof. Thus it is analogous to an *invariant* property required by proof rules for an iteration constructs in a Floyd-Hoare logic for an imperative programming language.

To show the soundness of P-logic we have proved the soundness of each of its proof rules relative to a standard semantics for Haskell98. This semantics, which is described elsewhere, has been manually derived by reference to the Haskell language definition [J^+99] and to numerous papers on specific aspects of the semantics. It has also been prototyped as a Haskell language interpreter and checked, although not systematically, for consistency with the Hugs interpreter, a mature implementation of Haskell98.

We have made no attempt to axiomatize properties of operators of predefined Haskell datatypes defined in the standard prelude. This includes the arithmetic operators, many operators on lists, IO operations While some of these operators are defined in Haskell as functions, many rely upon operations supported directly by hardware platforms or by an operating system. These operations are not particular to Haskell. The approach taken in defining P-logic is that properties of operations that are not Haskell-specific should be supported, insofar as is possible, by efficient, model-based decision procedures that may be invoked by a proof-search or proof-checking engine for P-logic. However, we have as yet no basis in experience with such a solution.

9.1 Future work

Much remains to be done to support property verification in P-logic with a robust and readily usable technology. Several tools must be constructed, as well as developing a content-specific software base. Most important among the tool-building tasks are:

- Build an extended Haskell front-end parser and type-checker that is capable of recognizing formulas of P-logic and translating them to terms in an abstract syntax. In Programatica, formula declarations and property assertions can be embedded in a Haskell program text, parsing and type-checking of formulas and assertions should be integrated with the front end of a Haskell translator to provide a uniform translation.

- Complete the encoding of rules and proof strategies in Stratego, adding specialized verification rules to P-logic as they are needed. We envision reaching toward an increasingly automated proof discovery system achieved through the accumulation of useful proof strategies that will be triggered as patterns of intermediate verification conditions are recognized in the course of developing a proof. Programmed strategies specify what rules may be tried to discharge proof obligations, and in what order alternative rules (and strategies) are tried.

  New strategies will be suggested by patterns of proofs discovered through pencil-and-paper proof attempts or with machine-aided, interactive proof construction. Once a strategy has been formalized and programmed, it can be tried automatically under programmed control.

- Adapt an interactive proof editor for use with P-logic. An interactive proof editor is analogous to an interactive programming environment. It must support browsing of data base of known proof rules and strategies, guided by its knowledge of the type and form of an assertion at the verification focus-point. An excellent model is provided by *Alfa* [Hal02], a proof editor for constructive type theory. *Alfa* uses the syntax of formulas and their type-theoretic structure to narrow the range of plausible selections among known terms that might be used in a next proof step. The principles successfully employed in *Alfa* may also be applicable to the design of a proof editor for P-logic.

Acknowledgments
Colleagues at the Pacific Software Research Center have helped to inform and improve this research with their comments. The author is particularly grateful to Bill Harrison and Jim Hook for their criticism and encouragement. The assistance provided by Francoise Bellegarde in carrying out experiments in programmed proof construction was invaluable. Any deficiencies that remain in the presentation are solely the responsibility of the author.

References

[Bvv+87] T. Brus, M.C.J.D. van Eekelen, M. van Leer, M.J. Plasmeijer, and H.P. Barendregt. CLEAN - a language for functional graph rewriting. In G. Kahn, editor, *Proc. of Conference on Functional Programming Languages and Computer Architecture (FPCA '87)*, number 274 in Lecture Notes in Computer Science, pages 364–384. Springer-Verlag, 1987.

[CE81] E. M. Clarke and A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In D. Kozen, editor, *Logics of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer Verlag, 1981.

[dvP01] Maarten de Mol, Marko van Eekelen, and Rinus Plasmeijer. Theorem proving for functional programmers. In T. Arts and M. Mohnen, editors, *Proceedings of the 13th International workshop on the Implementation of Functional Languages, IFL'01*, pages 99–118, September 2001.

[EC82] E. A. Emerson and E. M. Clarke. Using branching time temporal logic to synthesize synchronizations skeletons. *Science of Computer Programming*, 2:241–266, 1982.

[Hal02] Thomas Hallgren. The proof editor Alfa. http://www.cs.chalmers.se/ hallgren/Alfa/, 2002.

[HJS01] M. Huth, R. Jadadeesan, and D. Schmidt. Modal transition systems: a foundation for three-valued program analysis. In *Proc. European Symposium on Programming 2001*, volume 2028 of *Lecture Notes in Computer Science*. Springer, 2001.

[J$^+$99] Simon Peyton Jones et al. Report on the programming language haskell 98. Technical report, URL: www.haskell.org/definition/, February 1999.

[Koz83] Dexter Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27(3):333–354, December 1983.

[Lam80] Leslie Lamport. Sometimes is sometimes "not never"–on the temporal logic of programs. In *Conference Record of the Seventh Annual ACM Symposium on Principles of Programming Languages*, pages 174–185. ACM Press, 1980.

[Lar89] K. Larsen. Modal specifications. In J. Sifakis, editor, *CAV'89*, number 407 in Lecture Notes in Computer Science, pages 232–246. Springer, 1989.

[Owe97] Olaf Owe. Partial logics reconsidered: A conservative approach. *Formal Aspects of Computing*, 5:208–223, 1997.

[Pla00] M. J. Plasmeijer. CLEAN: a programming environment based on term graph rewriting. In A. Corradini and U. Montanari, editors, *Proc. of Joint COMPUGRAPH/SEMAGRAPH Workshop on Graph Rewriting and Computation (SEGRAGRA'95)*, volume 2 of *Electronic Notes in Theoretical Computer Science*, pages 233–240. Elsevier, 2000.

[Pnu77] A. Pnueli. The temporal logic of programs. In *Proc. Eighteenth IEEE Symposium on Foundations of Computer Science*, pages 46–57. IEEE Computer Society Press, 1977.

[Tho89] Simon Thompson. A Logic for Miranda. *Formal Aspects of Computing*, 1, July 1989.

[Tho95] Simon Thompson. A Logic for Miranda, Revisited. *Formal Aspects of Computing*, 7, March 1995.

[VeAB98] Eelco Visser and Zine el Abidine Benaissa. A core language for rewriting. In Claude Kirchner and Helene Kirchner, editors, *Second International Workshop on Rewriting Logic and its Applications (WRLA'98)*, Electronic Notes in Theoretical Computer Science. Elsevier, September 1998.

[Vis99] Eelco Visser. A bootstrapped compiler for strategies. In G. Gramlich, H. Kirchner, and F. Pfenning, editors, *Strategies in Automated Deduction (STRATEGIES'99)*, pages 78–83, July 1999.

[Vis01a] Eelco Visser. Scoped dynamic rewrite rules. In Mark van den Brand and Rakesh Verma, editors, *Rule Based Programming (RULE'01)*, volume 59/4 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, September 2001.

[Vis01b] Eelco Visser. Stratego: A language for program transformation based on rewriting strategies. System description of Stratego 0.5. In A. Middeldorp, editor, *Rewriting Techniques and Applications (RTA'01)*, volume 2051 of *Lecture Notes in Computer Science*, pages 357–361. Springer-Verlag, May 2001.

Appendix: Some Properties of Spook

In this Appendix, we illustrate the use of *P*-logic to formalize several properties relevant to Spook code. None of the assertions stated here has yet been proven, nor have the property formulas been type-checked.

```
-- Copyright (c) OHSU, 2002
module Schedule where

-----------------------------------------------------------------------
-- A trial run at stating the property that the scheduler, by
-- reshuffling the system level activities, may reshuffle the
-- top level activities, and the top level outputs. The individual
-- outputs will not be changed, but the order of the outputs may
-- be changed.
--
-- Something to watch out for. We want to assert the property
-- for both finite and infinite lists, and even for
-- lists that may be defined up to some point, but have
-- undefined tails after that point.
-----------------------------------------------------------------------

-- Haskell imports:
-- import qualified FiniteMap as FM
-- Spook imports:

import qualified FracturedState as FS

-- bigDot is an operator to compose a bunch of deltas

bigDot :: [FS.FSDelta k c] -> FS.FSDelta k c
bigDot = foldl (.) id

-- The important thing about the activity type a is that
-- it be an Ord type.

data (Enum a) => Activity a = Activity a

-- The output type from the system half

data Output b = Output b
data ActivityData k c =
    ActivityData
    {
    adDelta  :: FS.FSDelta k c,
    adLimits :: FS.FST k
    }
```

```
-- Some of the activities are top level, i.e. they come from
-- the user half to the system half. Others are not top level,
-- they are internal messages that flow to and from the system
-- half to other components of the kernel, in support of a top
-- level activity that came from the user.

topLevelActivity :: ActivityData k c -> Bool

-- For the property definitions given here, details of
-- the topLevel predicate are unimportant.

topLevelActivity = undefined

-- There is also a top level filter on the outputs.

topLevelOutput :: Output b -> Bool

-- These details are also unimportant here.

topLevelOutput = undefined

-- An activity is "refined" by the system half into a series
-- of activities, one of which is the original top level activity.
-- AN IMPORTANT SIMPLIFICATION (FOR NOW):
-- To simplify matters for the moment, we shall assume that a given
-- top level input always generates the same sequence of messages,
-- whereas in fact, the sequence of messages generated will depend upon
-- the state of the system at the time an activity arrives. We
-- will account for this complication later. If the complication were
-- propertly accounted for, then FS.FS k c would be one of the
-- paramaters to the function, refine.

refine :: Activity a -> [Activity a]

-- Exactly how the refinement is done is not germane to the
-- properties stated here

refine = undefined

-- Given the simplified refine, it is simple to extend it to
-- refine a list of top level activities, turning it into
-- a list of lists including both top-level and internal system activities.

refineStar :: [Activity a] -> [[Activity a]]
refineStar = map refine
```

```
-- This is implemented as a table lookup in Spook.
-- Here, even if it is undefined, the theorems should hold.

activityData :: Activity a -> ActivityData k c
activityData = undefined

-- An abstract view of what the system half does.
-- It takes a current value
-- of the fractured state, an input activity, and produces a new
-- value of the fractured state and an output. I have chosen to
-- produce a fractured state delta in the output. This is a function which,
-- when applied to the input fractured state, will produce the final
-- fractured state.

systemHalf ::
    (Ord k) =>
    FS.FS k c ->
    Activity a ->
    (FS.FSDelta k c, Output b)
systemHalf _fs _act =
  undefined

-- Define a fractured state delta that performs no change

nullFSDelta :: FS.FSDelta k c
nullFSDelta = id

-- Now extend systemHalf to work on a list of inputs, and
-- process them in the same order as the list. This version of
-- the system half can be run on a shuffled list of activities
-- produced by the refinement function.

systemHalfStar ::
    (Ord k) =>
    FS.FS k c ->
    [Activity a] ->
    (FS.FSDelta k c, [Output b])
systemHalfStar _fs [] = (nullFSDelta, [])
systemHalfStar fs (a:as) =
  let (delta, b) = systemHalf fs a
      (delta', bs) = systemHalfStar (delta fs) as
  in (delta' . delta, b : bs)

-- The schedule is the abstraction of the executive portion of the
-- system half. It takes the list of system level activities, and
-- shuffles them in an attempt to run things sooner.

schedule :: [[Activity a]] -> [Activity a]

-- Details of the scheduler are not important for the properties
-- stated here.

schedule = undefined

-- A run of the system half on a list of activities will produce
-- a whole bunch of deltas, and a whole bunch of outputs. The user
-- level summary is the composition of all the deltas, and the
-- list of top level outputs.

summarize ::
    [(FS.FSDelta k c, Output b)] ->
    (FS.FSDelta k c, [Output b])
summarize dbs =
  (bigDot (map fst dbs),                -- Compose the deltas
   filter topLevelOutput (map snd dbs)) -- Summarize the outputs

{-
---------------------------------------------------------------------
-- Now to state the properties. This section is surrounded by comment
-- braces, since we cannot parse the properties yet.
---------------------------------------------------------------------
-- To state the properties, we need to assume of the Haskell catenation
-- operator, ++, only that it induces a monoid on lists.  To prove assertions
-- that rely on ++, I expect we shall need its list homomorphism properties,
-- as expressed by the equations that define it as a Haskell function.
-- Otherwise, I see no way to eliminate some instances of existential
-- quantifiers. However, no proofs have yet been attempted. -- RBK, 2/15/2002

property Monoid = [| op, id | All x. All y. All z.
                                (x `op` y) `op` z === x `op` (y `op` z) /\
                                All x. x `op` id === x /\
                                All x. id `op` x === x
                  |]

assert Monoid (++) []

-- The property of finiteness of a list.

property FiniteList = Lfp X. [] \/ (Univ:$X)
```

```
-- Insert is a ternary predicate on lists.
-- It defines a relation between an element, 'a',  and two lists that
-- differ only in that the second list has an 'a' inserted somewhere.
-- One possibility is that the 'a' is put on the front of the list;
-- another possibility is that it was put somewhere in the tail.
-- This property will hold when the 'a' can be found in a
-- finite prefix of the second list, even though its tail
-- after the 'a' is found might be undefined.

property Insert =
  Lfp X. [| a, xs, ys | ys === a:xs
                        \/ (Exists x, xs', ys'. xs === x:xs' /\
                                                ys === x:ys' /\
                                                X a xs' ys') |]
-- Permutation is a binary relation on lists.
-- It relates lists that are permutations of one another.
-- For infinite lists, as with finite lists, this means that
-- there is a bijection between the two lists.

property Permutation =
  Gfp X. [| xs, ys | xs===ys \/
                     (Exists x. Exists xs'. xs===x:xs' /\
                         Exists ys'. Insert x ys' ys /\
                                     xs' `X` ys')
         |]

-- Embedded is a binary relation on lists.
-- It is the property that one list is embedded as a sublist of another.
-- This means that every element of the first list is an element
-- of the second list, and that these elements occur in the same order in
-- both. However, the embedded elements may appear interspersed with
-- other elements in the second list.
-- We only require this property when the first list is finite.
-- The second list could be infinite, and its tail (after all the
-- elements of the first list have been discovered) could be
-- undefined.

property Embedded =
  Lfp X.
     [| xs, ys | xs === [] \/
                 Exists x. Exists xs'. xs === x:xs' /\
                   Exists prefix, suffix, prefix'.
                     ys === prefix ++ suffix /\
                     prefix ::: Finitelist /\
                     Insert x prefix' prefix /\
                     xs' `X` suffix
     |]

-- EmbedsAll is a binary relation over types [[b]], [b].  It characterizes
-- the property that every element of the first list is embedded in the
-- second.
-- NOTE:  This property does not ensure that embeddings have unique inverse
-- projections.  For example, the following holds:
--    EmbedsAll [[a,b,d][c,d,f]] [a,b,c,b,c,d,e,f]

property EmbedsAll =
  Gfp X. [| xss, ys | xss === [] \/
                      Exists xs. Exists xss'. xss === xs:xss' /\
                         xs `Embedded` ys /\
                         xss' `X` ys |]

-- Flattened is a binary predicate on types [[b]], [b] which states
-- that the second list is equal to the iterated catenation of the
-- components of the first.  It does not assume finiteness of the
-- related lists, but only of the component lists of the first.

Flattened =
   Gfp X. [| xss, ys | (xss === [] /\ ys === []) \/
                       Exists xs. Exists xss'. xss === xs:xss' /\
                         xs ::: FiniteList /\
                         Exists ys'. ys === xs ++ ys' /\ xss' `X` ys'
          |]

-- Shuffle is a binary relation over types [[b]], [b].  It characterizes
-- the property that the second list consists only of embeddings of
-- elements from the first list.
-- This property can relate infinite lists, including the case
-- where both lists have (correspondingly) undefined tails.

property Shuffle = [| xss, ys | xss `EmbedsAll` ys /\
                                All xs'. xss `Flattened` xs' ==>
                                         xs' `Permutation` ys |]

-- NonOverlapped is a ternary property asserting that in an
-- embedding of two lists into a third list, the two do not
-- overlap. Only the two embedded lists need be finite for Spook.
-- The property stated here relies on the finiteness of the first
-- of the embedded lists, because it must be
-- embedded into a (finite) prefix of c.

property NonOverlapped =
  [| a, b, c | Exists cprefix, csuffix. c === cprefix ++ csuffix /\
                                          cprefix ::: FiniteList /\
                                          a `Embedded` cprefix /\
                                          b `Embedded` csuffix
  |]

-- NonConflicting is a binary property on lists of system actions.
-- It states that a two sequences of system level actions do not
-- conflict with each other. This property is only required of
-- finite lists.  FS.nonConflict is used as a binary property on lists of
-- activities and is exported by the FracturedState module.  However,
-- it may actually turn out to be a function of type [b] -> [b] -> Bool.

property NonConflicting =
         [| (a, b) | FS.nonConflict (map (adDelta . activityData) a)
                                    (map (adDelta . activityData) b) |]

-- NonOverlappedIfConflict is a ternary property relating lists.
-- It states that the embeddings of two lists of system activities
-- do not overlap in case they conflict with each other. This property
-- assumes only that each of the embedded lists is finite.

property NonOverlappedIfConflict =
  [| a, b, c |  NonConflicting a b \/
                Exists initc, lastc. c === initc ++ lastc  /\
                                     initc ::: FiniteList /\
                                     a `Embedded` initc /\
                                     b `Embedded` lastc
  |]

-- NonOverlapListConflictWith is a ternary property on lists.
-- It asserts that the embedding of a list 'a' in 'c' does not overlap
-- with an embedding in 'c' of any list in 'bs' with which 'a' is in conflict.
-- The list of lists 'bs' could be infinite.

property NonOverlapListConflictWith =
  Lfp X. [| a, bs, c | bs ::: [] \/
                       Exists b. Exists bs'.
                           bs === b:bs' /\
                           NonOverlappedIfConflict a b c /\
                           X a bs' c
         |]

-- NonOverlapListConflict is a binary property relating a list of lists
-- of system activities to a flat list of system activities. It asserts that
-- each list of the first is embedded in the second and that the embeddings
-- do not overlap with an embedding of any list with which there is conflict.
```

```
property NonOverlapListConflict =
  Lfp X.  [| as, c | as ::: [] \/
                     Exists a. Exists as'.
                        NonOverlapListConflictWith a as c /\
                        as `X` c
          |]


-------------------------------------------------------------------
-- Now to use these properties in assertions about the Spook
-- abstractions.
-------------------------------------------------------------------

-- The refine operation embeds the top level input into the
-- resulting list of activities.  `Elem` is a binary property
-- satisfied if its first argument is a member of the list given
-- as its second argument.  It is a lifting of the Haskell function
-- `elem` to a predicate.

assert All (x :: Activity a). x `Elem` (refine x)


-- The schedule is just a fancy permutation.
-- This assertion should be valid when 'as' is infinite, and when
-- 'as' has an undefined tail.

assert All (ass :: [[Activity a]]). All (as :: [Activity a]).
   ass `Flattened` as => (schedule ass) `Permutation` as

-- When a list of top level inputs is refined into the system
-- level inputs, and then the scheduler reshuffles the system level
-- activities, the resulting list of system level activities does
-- not have any overlapping activities that conflict with each other.
-- This property should work on infinite lists, including infinite
-- lists that have undefined tails.

assert All (as :: [[Activity a]]).
   as `NonOverlapListConflict` ((schedule . refineStar) as)

-- Now assert that when the system is run, the scheduling does not
-- affect the outputs of the system, except possibly reordering them.
-- This assertion should work for infinite lists, including those
-- with undefined tails.

assert All (ass :: [[Activity a]]).
```

```
         All (as :: [Activity a]).
           All (ss :: [Activity a]).
             (ass ‘Flattened‘ as /\
              ss === (schedule . refineStar) ass) ==>
             (summarize . systemHalfStar) as ===
             (summarize . systemHalfStar) ss

-}
```