Automated soundness checking of a programming logic for Haskell

Richard B. Kieburtz

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

Abstract

P-logic is a verification logic for the programming language Haskell. Inference rules are expressed in sequent calculus for each of the term constructs of Haskell. Validating soundness of these rules is an essential task. Most rules of P-logic are polymorphic, independent of Haskell's type classes. The paper develops a parametricity principle for predicates of P-logic, which justifies checking soundness of polymorphic rules at a particular type instance. By implementing the Haskell semantics with a meta-circular interpreter, automated model-checking has been used to validate soundness of the polymorphic inference rules of P-logic.

1 Introduction

P-logic is a verification logic for the programming language Haskell. Its term language is the set of Haskell terms that are syntactically well-formed and well-typed in a context of module, class, type and expression declarations.

As a verification logic, it is comprised of inference rules, here expressed in sequent calculus style, that characterize the functional behavior of Haskell terms. However, since Haskell is not defined in terms of *P*-logic, an immediate question is: how can we be sure that the inferences made in the logic are coherent with the abstract semantics that actually defines Haskell? In other words, how do we know that *P*-logic is sound?

Generally, soundness of a logic is shown by exhibiting a model for it, i.e. a non-trivial interpretation of the logic. The need to provide models to assure sound logical reasoning about rich, programming calculi provided the original motivation for developing denotational semantics.

P-logic is unusually interesting as a verification logic because it is able to express a critical properties of programs in a non-strict language, such as whether a term in context necessarily denotes a well-defined value. With the ability to define predicates recursively, either finiteness or unboundedness of data structures is expressible in *P*-logic.

Given a standard model for Haskell, to show that it is actually a model for *P*-logic we must prove that each of the inference rules of the logic is valid for the model. More specifically, a soundness proof of a rule consists in showing that under every type-compatible assignment of values to the free variables occurring in the rule, its consequence is true of the model whenever all of its hypotheses are true. Our use of the phrase "free variables" refers here to both term and predicate variables.

The inference rules of *P*-logic comprise a *core* set, characterizing the definitional and expression constructions of Haskell, plus a set of algebraic laws for the interpreted operators of arithmetic data types, monads and the derived operators of certain classes. We are concerned here only with the core set of rules. The rules in the core set, since they do not attempt to axiomatize specific algebras, are polymorphic.

Since the core logic is polymorphic, it is reasonable to expect a parametricity result to hold. We expect that if a rule is valid when interpreted at any specific type, it must be valid at all types. With such a result, which we prove in Section 3, a soundness proof of an inference rule can be based upon an interpretation of the rule at a single type, which may be chosen to be finite. Checking the validity of a rule in all interpretations in a finite type becomes a finite model-checking problem.

In the remainder of the paper, the ideas sketched here are developed more formally. In Section 2, we give a brief introduction to *P*-logic and its semantics, which have been more fully defined elsewhere [6]. The meta-theory of parametricity for *P*-logic is developed in Section 3, using logical relations.

A meta-circular interpreter provides a denotational model of Haskell [4] which we have used in automating model-checking of inference rules of *P*-logic. Model-checking is described in Section 4. Section 5 summarizes conclusions and directions for future work.

2 A brief introduction to P-logic

P-logic is designed to express property assertions of Haskell expressions, utilizing the context of a Haskell program to provide bindings for free term variables that may occur in an expression. The term language of *P*-logic is Haskell98. The formula language is a predicate calculus with interpreted equality, extended into a mu-calculus which allows least and greatest fixed-point formulas to define predicates, and equipped with a modality for expression evaluation.

The atomic propositional forms of *P*-logic either assert term equality, $t_1 \texttt{===} t_2$, or that a term has a property expressed as a unary predicate, $t ::: P$, or more generally, that a k-ary predicate expresses a property of k terms. The symbol (:::) expresses a (unary) property assertion, in analogy to the symbol (::) that expresses a typing assertion in Haskell. In this paper we shall consider only those formulas that express unary predicates, referring the reader to a more complete definition of the logic [6] for the treatment of multi-place predicate formulas, including equality.

Informally, we intend that an assertion $t ::: P$ should be true in a given context if evaluating t by Haskell's denotational semantics results in a value 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 predicate are sensible.

We say that a predicate, P, is *weakly* satisfied by an expression M if the denotation of M belongs to the set defined by the interpretation of P. It is *strongly* satisfied if in addition, the denotation of M is non-bottom value. An otherwise weak assertion context can be explicitly strengthened by prefixing the symbol ($) to a predicate formula. A strengthened predicate can only be strongly satisfied.

2.1 Predicate formulas

There are two atomic predicates in *P*-logic

- Univ is the universal predicate satisfied by all terms,
- UnDef is the predicate of undefinedness, satisfied only by terms which denote $\bot$.

There are five 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. Predicates formed in this way are called *term congruences*. For example, in the context of a formula, the list constructor combines two predicates, P and Q, into a new predicate formula, $(P : Q)$. This formula is satisfied by a Haskell expression that evaluates to a 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 list constructor allows undefined values as its arguments.
- The "arrow" predicate constructor is used to compose formulas that express properties of function-typed expressions. A formula $P \rightarrow Q$ is satisfied by a function if when the function is applied to an argument that satisfies P, the resulting application satisfies Q.
- A least or greatest fixpoint binder may bind a predicate variable in a prefix of a predicate formula. The μ and ν binders of the mu-calculus, but are written as Lfp and Gfp in formulas of *P*-logic.
- The propositional connectives "$\wedge$" and "$\vee$" are lifted to predicate constructors, with meanings defined by

$$t ::: (P \wedge Q) \equiv t ::: P \wedge t ::: Q$$
$$t ::: (P \vee Q) \equiv t ::: P \vee t ::: Q$$

2.2 The modality of Haskell terms

When used in conjunction, lifted connectives, term congruences and fixpoint binders allow detailed properties of Haskell expressions to be formulated in *P*-logic. For example, a unary predicate asserting that an expression of type *List a* denotes a finite list can be defined as

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

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 empty list. The second disjunct is satisfied by a term denoting a constructed list whose tail is non-bottom and satisfies the *Finite_list* property.

To characterize lists for which every listed element satisfies a common property, Q, we can declare

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

in which a predicate, Q, is a parameter of the declaration.

The property *All_elts* includes both finite and infinite lists. This property definition differs from that of *Finite_list* in two important details: (1) the binding operator in the formula is Gfp rather than Lfp and (2) the predicate variable, X, is unstrengthened.

2.3 Inference rules in *P*-logic

Rules in *P*-logic are shown in Figure 1 in a sequent calculus style[1]. Unlike a natural deduction style, in which complementary introduction and elimination rules are given for each construction of the term language, there are only introduction rules in sequent calculus style. However, a property introduction may occur either on the right of the turnstile symbol, as a conclusion, or to the left, as an assumption. Assumption introductions play a similar role in the sequent calculus as do elimination rules in natural deduction style. The sequent calculus is well suited to a goal-directed verification logic, as each rule specifies verification conditions for a property matching its consequent.

2.3.1 Fixed-point properties of a recursive function definition

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 definable 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.

2.3.2 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.

In rule (12) for l.f.p. properties, X is a predicate variable that may occur in H but does not occur in P_1 or P_2 and $X \notin FV(\Gamma)$.

The first antecedent clause of (12) provides a base case for induction. The assertion m ::: 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 of (12) 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 === t_m, is added to the context. The constraint, $P \Rightarrow P_1 \vee P_2$, is necessary to ensure soundness of the rule. For completeness, we usually want the additional condition, $P_1 \vee P_2 \Rightarrow P$.

2.3.3 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.

Rule (13) concludes a fixed-point property of a recursively-defined term. The predicate variable X may occur in H but not in P and $X \notin FV(\Gamma)$. The second antecedent clause asserts that when an t_m strongly 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 base case for induction is the first antecedent, $\Gamma \vdash t_m$ ::: $\$H[\mathsf{Univ}/X]$.

2.4 Semantics of *P*-logic

The semantic models we shall consider for Haskell are based upon the universe of typed ideals of MacQueen, Plotkin and Sethi [7, 1]. In an ideal model, the elements of a c.p.o. semantic domain constitute a single, untyped universe. Ideals are downward-closed sets which contain the limits of their directed subsets. Ideals have non-empty intersections; in particular, the bottom element of a pointed, c.p.o. universe belongs to each of its ideals. The types of Haskell expressions are naturally modeled as ideals, since every Haskell type has a bottom element. The limit points of chains in an ideal that models a type represent the "values" of the type. Not all ideals are associated with types, however.

Ideals also provide satisfactory models for the admissible predicates of *P*-logic, which refine the types of Haskell. As we shall see, *P*-logic also specifies a strong modality for predicates, in which the interpretation of the predicate "cuts

[1]The set of rules given in Figure 1 is incomplete. In particular, no structural rules to justify weaker conclusions or stronger assumptions are shown.

[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.

$$\frac{\Gamma \vdash M ::: P \qquad \Gamma \vdash N ::: Q}{\Gamma \vdash M, N \ ::: \ \$(P,\, Q)} \tag{1}$$

$$\frac{M ::: P \vdash \Delta}{(M, N) ::: (P, \mathsf{Univ}) \vdash \Delta} \tag{2}$$

$$\frac{N ::: Q \vdash \Delta}{(M, N) ::: (\mathsf{Univ}, Q) \vdash \Delta} \tag{3}$$

$$\frac{\Gamma \vdash M ::: \$(P,\, \mathsf{Univ})}{\Gamma \vdash \mathsf{fst}\ M \ ::: \ P} \qquad \frac{\Gamma \vdash M ::: \$(\mathsf{Univ},\, Q)}{\Gamma \vdash \mathsf{snd}\ M \ ::: \ Q} \tag{4}$$

$$\frac{\Gamma,\, x ::: P \vdash M ::: Q}{\Gamma \vdash (\lambda x \ \texttt{->}\ M) \ ::: \ \$(P \to Q)} \tag{5}$$

$$\frac{\Gamma \vdash N ::: P \qquad \Gamma,\, M\ N ::: Q \vdash \Delta}{\Gamma,\, M ::: \$(P \to Q) \vdash \Delta} \tag{6}$$

$$\frac{\Gamma \vdash M ::: \$(P \to Q) \qquad \Gamma \vdash N ::: P}{\Gamma \vdash M\ N \ ::: \ Q} \tag{7}$$

$$\frac{M ::: P \to Q \vdash \Delta}{x ::: P,\, M\ x ::: Q \vdash \Delta} \quad \text{(variable } x \notin FV(M)\text{)} \tag{8}$$

$$\frac{\Gamma \vdash M_1 ::: P_1 \ \cdots \ \Gamma \vdash M_k ::: P_k}{\Gamma \vdash C^{(k)}\ M_1 \ldots M_k ::: \$C^{(k)}\ P_1 \ldots P_k} \quad (k \geq 0) \tag{9}$$

$$\frac{M_i ::: P_i \vdash \Delta}{C^{(k)}\ M_1 \ldots M_i \ldots M_k ::: C^{(k)}\ \mathsf{Univ} \ldots P_i \ldots \mathsf{Univ} \vdash \Delta} \tag{10}$$

$$\frac{\Gamma \vdash M_1 ::: P_1 \ldots \Gamma \vdash M_j ::: \$P_j \ldots \Gamma \vdash M_k ::: P_k}{\Gamma \vdash C^{(k)}\ M_1 \ldots M_j \ldots M_k ::: \$C^{(k)}\ P_1 \ldots \$P_j \ldots P_k} \quad (1 \leq j \leq k) \tag{11}$$

$$\frac{\begin{array}{c}\Gamma,\, m ::: \mathsf{Univ} \ \vdash t_m ::: \$(P_1 \to H) \\ \Gamma,\, m ::: \$(P \to X) \vdash t_m ::: \$(P_2 \to H)\end{array}}{\Gamma,\, m \texttt{===} t_m \vdash m ::: \$(P \to \mathsf{Lfp}\, X \bullet H)} \quad \text{where } m \notin FV(\Gamma) \text{ and } P \Rightarrow P_1 \vee P_2 \tag{12}$$

$$\frac{\Gamma \vdash t_m ::: \$H[\mathsf{Univ}/X] \qquad \Gamma,\, t_m ::: \$H \vdash m ::: X}{\Gamma,\, m \texttt{===} t_m \vdash m ::: \mathsf{Gfp}\, X \bullet H} \tag{13}$$

Figure 1. Inference rules of P-logic

off" the bottom element of its ideal model. A strong predicate is satisfied only by expressions whose denotation is not $\bot$.

2.5 A semantic interpretation of P-logic

Let $\mathcal{A}[\![\ \_\ ]\!] :: (\mathit{Term} \times \mathit{Type}) \to \mathit{Env} \to \mathcal{A}$ be a meaning function that maps every well-typed Haskell expression to its denotation in a domain $\mathcal{A}$, where $\mathit{Env} = \mathit{Var} \times \mathit{Type} \to \mathcal{A}$. When the domain of interpretation is unambiguous, as when we are only talking about a single domain, the domain identifier will be omitted.

We use the following notation to distinguish terms and formulas from their meanings:

$\mathcal{A}\lceil\tau\rceil_\bot$	is the ideal containing interpretations of terms of type τ
$\mathcal{A}\lceil\tau\rceil$	is the set of interpretations of terms of type τ, excluding $\bot_\mathcal{A}$ (i.e., $\mathcal{A}\lceil\tau\rceil = \mathcal{A}\lceil\tau\rceil_\bot \setminus \{\bot_\mathcal{A}\}$)

The meaning of a term constant at a type, τ, is

$$\mathcal{A}[\![C, \tau]\!]\eta = C_{\mathcal{A},\tau} \in \mathcal{A}\lceil\tau\rceil_\bot$$

Formulas will be interpreted as characteristic predicates of sets (posets) in an abstract domain for Haskell's semantics. The meaning of a predicate formula, P, at a type *Pred* τ, is denoted as

$$\mathcal{A}[\![P]\!]^{\mathit{Pred}\ \tau} \subseteq \ \mathcal{A}\lceil\tau\rceil_\bot$$

The interpretation of a strong predicate is

$$\mathcal{A}[\![\$P]\!]^{\mathit{Pred}\ \tau} = \mathcal{A}[\![P]\!]^{\mathit{Pred}\ \tau} \setminus \{\bot_\mathcal{A}\}$$

2.5.1 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:

$$[\![\mathsf{Univ}]\!]^{\mathit{Pred}\ \tau} = \lceil\tau\rceil_\bot \qquad [\![\mathsf{UnDef}]\!]^{\mathit{Pred}\ \tau} = \{\bot_\tau\}$$
$$[\![\$\mathsf{Univ}]\!]^{\mathit{Pred}\ \tau} = \lceil\tau\rceil \qquad [\![\$\mathsf{UnDef}]\!]^{\mathit{Pred}\ \tau} = \{\,\}$$

2.5.2 Term congruence predicates

The meaning of a term congruence predicate formed with a k-ary constructor, $C \in \Sigma_k^\tau$, is

$$\begin{array}{l}((C, (\tau_1, \ldots, \tau_k)) \in \Sigma_k^\tau) \ \Rightarrow \ [\![C\ P_1 \cdots P_k]\!]^{\mathit{Pred}\ \tau} = \\ \quad \{C_{\lceil\tau\rceil}\ t_1 \cdots t_k \ \mid \ t_1 \in [\![P_1]\!]^{\mathit{Pred}\ \tau_1} \ \wedge \ldots \\ \qquad\qquad \wedge\, t_k \in [\![P_k]\!]^{\mathit{Pred}\ \tau_k}\} \ \cup \ \{\bot\}\end{array}$$

The above interpretation is for a non-strict datatype constructor.

2.5.3 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.

$$\begin{array}{l}[\![P \to Q]\!]^{\mathit{Pred}\ \tau_1 \to \tau_2} = \\ \{f \in \lceil\tau_1\rceil_\bot \to \lceil\tau_2\rceil_\bot \ \mid \ \forall x \in [\![P]\!]^{\mathit{Pred}\ \tau_1} \bullet f\ x \in [\![Q]\!]^{\mathit{Pred}\ \tau_2}\} \\ \cup \{\bot\}\end{array}$$

where the function space is that of continuous functions from $\lceil\tau_1\rceil_\bot$ to $\lceil\tau_2\rceil_\bot$.

2.5.4 Predicate conjunction and disjunction

$$[\![P_1 \wedge P_2]\!]^{Pred\ \tau} = [\![P_1]\!]^{Pred\ \tau} \bigcap [\![P_2]\!]^{Pred\ \tau}$$
$$[\![P_1 \vee P_2]\!]^{Pred\ \tau} = [\![P_1]\!]^{Pred\ \tau} \bigcup [\![P_2]\!]^{Pred\ \tau}$$

2.5.5 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.

$$[\![\mathsf{Lfp}X.\,H]\!]^{Pred\ \tau} = \bigcup_{j=0}^{\infty} [\![H^j]\!]^{Pred\ \tau}$$
where $H^0 = \mathsf{UnDef}$
$H^{j+1} = H[H^j/X]$

$$[\![\mathsf{Gfp}X.\,H]\!]^{Pred\ \tau} = \bigcap_{j=0}^{\infty} [\![H^j]\!]^{Pred\ \tau}$$
where $H^0 = \mathsf{Univ}$
$H^{j+1} = H[H^j/X]$

3 Logical relations and parametricity

Logical relations were introduced into programming language theory by Mitchell and Meyer [9] to formalize properties of the second-order lambda calculus. They have since been used by many authors to derive parametricity properties of polymorphic functions as well as representation independence [11, 3, 2]. Intuitively, a logical relation is a type-indexed family of relations between (possibly different) semantic representations which preserves the algebraic behavior induced by modeling the calculus. Here, we consider logical relations for a calculus with the type system of Haskell, in which type variables may universally quantified, but only in a global context.

3.1 Logical relations over domains

Logical relations were first developed for strongly normalizing applicative structures that provide semantics for typed lambda calculi without recursion. Logical relation can be extended to full, continuous hierarchies of c.p.o.'s that provide semantics for lambda-*fix* calculi. These relations must respect the partial order structure of domains.

Polymorphism is implied by universal quantification over a type variable. The meaning of a polymorphic expression is the intersection of its meanings taken at all instances of the quantified type variable, $[\![M]\!]^{\forall\alpha.\sigma} = \bigcap_{\tau::type} [\![M]\!]^{\sigma[\tau/\alpha]}$. Clearly, the meaning of every polymorphic expression includes $\perp$, as it belongs to the meaning of the expression at every specific type.

Parametricity is consequent to the interpretation of type quantification. At a polymorphic type,

$$\mathcal{R}^{\forall\alpha.\sigma} = \bigcap_{\tau::type} \mathcal{R}^{\sigma[\tau/\alpha]}$$

This interpretation has the consequence that meanings related by $\mathcal{R}^{\forall\alpha.\sigma}$ cannot depend upon constants of any particular type that might substitute for α.

Corollary 3.1: Suppose $\mathcal{A}$ and $\mathcal{B}$ are pointed c.p.o. domains and let $\mathcal{R} \subseteq \mathcal{A} \times \mathcal{B}$ be logical. For every pointed type, σ, $\mathcal{R}^{\sigma}(\perp_{\mathcal{A}}, \perp_{\mathcal{B}})$.

3.2 Models for *P*-logic

A model for *P*-logic extends a Haskell model by providing interpretations for predicate constants and predicate constructors. We wish the meanings of predicates to refine the meanings of types and furthermore, to give meanings to predicates defined as least or greatest fixed points of predicate constructors. The meaning of a *Pred* σ-typed predicate in *P*-logic can be defined as a characteristic predicate over the domain that interprets the corresponding Haskell term type, σ. As was mentioned briefly in Section 2, ideal models are also appropriate for predicates.

A type-indexed family of unary predicates, $\mathcal{P}$, over an applicative domain of c.p.o.s, $\mathcal{A}$, is *directed-complete* if it is closed under limits of directed sets, at every type; *pointed* if $P^{\sigma}(\perp)$ at every type, σ. Directed-complete (resp. pointed) predicates are closed with respect to the operations of predicate disjunction and predicate conjunction. A predicate arrow, $P \rightarrow Q$, constructed from directed complete predicates, P and Q, is also directed complete, as it is comprised of the continuous functions from a domain satisfying P to a domain that satisfies Q. Continuous functions are just those which preserve order and respect limits of directed sets.

Definition3.1: A (unary) predicate formula, $\mathcal{P}$ is *logical* iff it is directed-complete and downward-closed.

A logical predicate is pointed at every pointed type, which for *P*-logic, is every type.

3.2.1 Polymorphic predicates

Notice that the type of a predicate is contravariant in the type of expressions that may satisfy it; i.e. *Pred* $\tau = \tau \rightarrow$ *Prop*. This has an important consequence; the meaning of a polymorphic predicate is the union of its meanings at all type instances: $[\![P]\!]^{\forall\alpha.Pred\ \sigma} = \bigcup_{\tau::type} [\![P]\!]^{Pred\ \sigma[\tau/\alpha]}$.

For example, the (polymorphic) meaning of the universal predicate, Univ, is $[\![\mathsf{Univ}]\!]^{\forall\alpha.Pred\ \alpha} = \bigcup_{\tau::type} [\![\mathsf{Univ}]\!]^{Pred\ \tau} = \bigcup_{\tau::type} \lceil\tau\rceil_{\perp}$, which is the entire semantic domain.

3.2.2 The strong modality

A *P*-logic predicate in the $-modality satisfies the definition of a logical predicate except that it is (specifically) unpointed. That is, its semantic interpretation is directed-complete and downward closed except that the bottom element of the domain in its type is removed from the interpretation. We say of such a predicate that it is *strongly logical*.

3.3 Predicate logical relations and the representation independence lemma

The Basic Lemma for logical relations establishes representation-independence of term reduction. To establish representation independence of predicate satisfaction in *P*-logic, it will suffice to show that observations restricted by predicate satisfaction are logical, i.e. that they satisfy the conditions of a logical relation.

More specifically, for a class of admissible predicates[3], two equivalently typed terms are related modulo observation by a predicate formula, in related term and predicate environments, if satisfaction of the proposition in any one model is logically equivalent to satisfaction in any other model.

Predicate formulas characterize the observations on which a relation is founded. A predicate variable occurring in the formula may be instantiated to any logical or strongly logical predicate of the type assigned to the variable, just as term variables may be instantiated by type-conforming valuations.

Since the strong modality of a predicate in *P*-logic restricts the satisfying interpretations of a term to non-bottom elements of the semantic domain in its type, it is possible to express relations that cannot be expressed by relations between unpredicated terms. Such relations lead to "free theorems"; for example, that there is a unique, non-bottom member of the type $\forall\alpha \bullet \alpha \to \alpha$. Using only relations between unpredicated terms, the strongest provable result for this type is that it contains only two members, namely the identity function and $\bot$.

3.4 Logical relations extend to models of predicate formulas

A logical relation between domains, $\mathcal{R} \subseteq \mathcal{A}\times\mathcal{B}$, induces a relation between powersets of the domains, $\mathcal{R}^{Pred\ -} \subseteq \wp(\mathcal{A}) \times \wp(\mathcal{B})$. This enables us to relate models of predicate formulas.

Definition 3.2:
If $\mathcal{R} \subseteq \mathcal{A} \times \mathcal{B}$ is a logical relation, predicates $P :: \mathit{Pred}\ \sigma$ and $Q :: \mathit{Pred}\ \sigma$ are related by $\mathcal{R}^{Pred\ \sigma} \subseteq \wp(\mathcal{A}) \times \wp(\mathcal{B})$ iff for all terms $s, t :: \sigma$, $\mathcal{R}^{\sigma}(\mathcal{A}[\![s]\!], \mathcal{B}[\![t]\!]) \Rightarrow (\mathcal{A}[\![s]\!]^{\sigma} \in \mathcal{A}[\![P]\!]^{Pred\ \sigma} \Leftrightarrow \mathcal{B}[\![t]\!]^{\sigma} \in \mathcal{B}[\![Q]\!]^{Pred\ \sigma})$.

[3]The formulas constructed inductively from the formation rules given in Section 2 are admissible, as no predicate variable occurs in a negated position.

Lemma 3.1: Let $\sigma,\ \sigma_1, \ldots, \sigma_n,\ \tau$ be pointed types and suppose $\mathcal{R} \subseteq \mathcal{A} \times \mathcal{B}$ is logical. Then

- $\mathcal{R}^{Pred\ \sigma}(\mathcal{A}[\![\mathsf{Univ}]\!], \mathcal{B}[\![\mathsf{Univ}]\!])$
- $\mathcal{R}^{Pred\ \sigma}(\mathcal{A}[\![\mathsf{UnDef}]\!], \mathcal{B}[\![\mathsf{UnDef}]\!])$
- $\mathcal{R}^{Pred\ \sigma}(\mathcal{A}[\![\$P]\!], \mathcal{B}[\![\$Q]\!]) \Leftrightarrow \mathcal{R}^{Pred\ \sigma}(\mathcal{A}[\![P]\!], \mathcal{B}[\![Q]\!])$
- $\mathcal{R}^{Pred\ \sigma}(\mathcal{A}[\![P_1]\!], \mathcal{B}[\![P_2]\!]) \wedge$
  $\mathcal{R}^{Pred\ (\sigma\to\tau)}(\mathcal{A}[\![P_1 \to Q_1]\!],\ \mathcal{B}[\![P_2 \to Q_2]\!]) \Rightarrow$
  $\mathcal{R}^{Pred\ \tau}(\mathcal{A}[\![Q_1]\!], \mathcal{B}[\![Q_2]\!])$
- $\mathcal{R}^{Pred\ \sigma}(\mathcal{A}[\![C^n P_1 \ldots P_n]\!], \mathcal{B}[\![C^n Q_1 \ldots Q_n]\!]) \Leftrightarrow$
  $\forall i \in 1\,..\,n \bullet \mathcal{R}^{Pred\ \sigma_i}(\mathcal{A}[\![\overline{P_i}]\!],\ \mathcal{B}[\![\overline{Q_i}]\!])$
  where $C^n :: \sigma_1 \times \cdots \times \sigma_n \to \sigma$ is an *n*-place constructor
  and $\overline{P_i} = \begin{cases} \$P_i & \text{if } C^{(n)} \text{ is strict in its } i^{th} \text{ argument} \\ P_i & \text{otherwise} \end{cases}$
  and $\overline{Q_i}$ is defined similarly
- $\mathcal{R}^{Pred\ \sigma}(\mathcal{A}[\![P_1]\!], \mathcal{B}[\![P_2]\!]) \wedge \mathcal{R}^{Pred\ \sigma}(\mathcal{A}[\![Q_1]\!],\ \mathcal{B}[\![Q_2]\!]) \Rightarrow$
  $\mathcal{R}^{Pred\ \sigma}(\mathcal{A}[\![P_1 \wedge Q_1]\!], \mathcal{B}[\![P_2 \wedge Q_2]\!])$
- $\mathcal{R}^{Pred\ \sigma}(\mathcal{A}[\![P_1]\!], \mathcal{B}[\![P_2]\!]) \vee \mathcal{R}^{Pred\ \sigma}(\mathcal{A}[\![Q_1]\!],\ \mathcal{B}[\![Q_2]\!]) \Rightarrow$
  $\mathcal{R}^{Pred\ \sigma}(\mathcal{A}[\![P_1 \vee Q_1]\!], \mathcal{B}[\![P_2 \vee Q_2]\!])$

Proof: The proof is by induction on the structure of non-recursive predicates. (See Appendix).

Corollary 3.1: Type generalization of relations between predicates.

$$\mathcal{R}^{\forall\alpha.Pred\ \sigma}(\mathcal{A}[\![P]\!], \mathcal{B}[\![Q]\!]) = \bigcup_{\tau::type} \mathcal{R}^{Pred\ \sigma[\tau/\alpha]}(\mathcal{A}[\![P]\!], \mathcal{B}[\![Q]\!])$$

3.4.1 Predicate environments

A predicate environment maps predicate variables to their meanings in a model. We write $LP(\mathcal{A})$ to designate the class of logical and strongly logical predicates over a semantic domain, $\mathcal{A}$. To express that predicate environments $\pi_a :: \mathit{PredVars} \to LP(\mathcal{A})$ and $\pi_b :: \mathit{PredVars} \to LP(\mathcal{B})$ respect a type environment, Γ, we write $\pi_a, \pi_b \models \Gamma$.

Definition 3.3: $\mathcal{R}^{\Gamma}(\pi_a, \pi_b) \equiv$
$\forall P \in \mathit{PredVars} \bullet \mathcal{R}^{\Gamma}(\pi_a(P), \pi_b(P))$, where $\pi_a, \pi_b \models \Gamma$

Relations between predicates can now be extended to predicate formulas in which there may be free occurrences of predicate variables. We write $\mathcal{R}^{Pred\ \sigma}(\mathcal{A}[\![P]\!]\pi_a,\ \mathcal{B}[\![P]\!]\pi_b)$ to express such a relationship.

3.4.2 Extending logical relations to recursively specified predicates

In a pointed applicative c.p.o. structure, $\mathcal{A}$, the semantics of a least fixed-point predicate formula is given by

$$\mathcal{A}[\![\mathsf{Lfp}\,\xi \bullet H]\!]^\sigma = \bigcup_{i=0}^{\infty} \mathcal{A}[\![H^i[\mathsf{UnDef}]]\!]^\sigma$$

where $H^0[P] = P$
$H^{i+1}[P] = H[H^i[P]/\xi]$.

The semantics of a greatest fixed-point formula is:

$$\mathcal{A}[\![\mathsf{Gfp}\,\xi \bullet H]\!]^\sigma = \bigcap_{i=0}^{\infty} \mathcal{A}[\![H^i[\mathsf{Univ}]]\!]^\sigma$$

Lemma 3.2 Let σ be a pointed type and suppose $\mathcal{R} \subseteq \mathcal{A}\times\mathcal{B}$ is logical. Then

$\mathcal{R}^{\mathit{Pred}\ \sigma}(\mathcal{A}[\![\mathsf{Lfp}\,\xi \bullet H]\!],\ \mathcal{B}[\![\mathsf{Lfp}\,\xi \bullet H]\!])$ and
$\mathcal{R}^{\mathit{Pred}\ \sigma}(\mathcal{A}[\![\mathsf{Gfp}\,\xi \bullet H]\!],\ \mathcal{B}[\![\mathsf{Lfp}\,\xi \bullet H]\!])$
iff $\exists P :: \mathit{Pred}\ \sigma \bullet \mathcal{R}^{\mathit{Pred}\ \sigma}(\mathcal{A}[\![P]\!],\ \mathcal{B}[\![P]\!]) \Rightarrow$
$\mathcal{R}^{\mathit{Pred}\ \sigma}(\mathcal{A}[\![H[P/\xi]]\!],\ \mathcal{B}[\![H[P/\xi]]\!])$

Proof: (See Appendix).

3.4.3 Parametricity for polymorphic predicates

The interpretation of predicates under type generalization yields a parametricity property that is dual to the property of parametricity for terms. Under type generalization, the meaning of a term can depend on no semantic element that is particular to any instance of a generalized type variable. This restricts the meanings assigned to terms of a polymorphic type, yielding unique elements in a set interpretation and unique upper bounds in a c.p.o. interpretation.

Dually, the meaning of a predicate can impose no constraint that is particular to an instance of a generalized type variable. Consequently, two predicates are related at a generalized type iff they are related at any particular instance of the type.

Theorem 3.1: Parametricity for predicates.

Let $\mathcal{R} \subseteq \mathcal{A} \times \mathcal{B}$ be a logical relation. Then the following are logically equivalent conditions for the induced relation of predicates P and Q at a generalized type:

$\mathcal{R}^{\forall\alpha.\mathit{Pred}\ \sigma}(\mathcal{A}[\![P]\!], \mathcal{B}[\![Q]\!])$ and
$\exists\tau :: \mathit{type} \bullet \mathcal{R}^{\mathit{Pred}\ \sigma[\tau/\alpha]}(\mathcal{A}[\![P]\!], \mathcal{B}[\![Q]\!])$

Proof: (See Appendix.)
□

A type-parametric property can be observed at any type instance of the polymorphic predicate that encodes the property. The predicate parametricity theorem formalizes this intuition. For example, the polymorphic predicate $\mathit{DefinedHead} = (\$\mathsf{Univ} : \mathsf{Univ}) :: \forall\alpha.\mathit{Pred}\ [\alpha]$ is satisfied (in any model) at the type $[\mathsf{Int}]$ by the meaning of the term [1,2,3] and at the type $[\mathsf{Char}]$ by the meaning of the term ('A' : *undefined*).

3.4.4 Satisfaction of a predicate formula is model-independent

A judgment that a typed term satisfies a predicate can now be given a semantic interpretation. In a model, $\mathcal{A}$, with value environment, η, and predicate environment, π, the meaning of the judgment form $\Gamma \rhd M :: \sigma ::: P$ is

$$\mathcal{A}[\![\Gamma \rhd M :: \sigma ::: P]\!]\,\eta\,\pi = \mathcal{A}[\![\Gamma \rhd M :: \sigma]\!]\eta \in \mathcal{A}[\![\Gamma \rhd P :: \mathit{Pred}\ \sigma]\!]\pi$$

We shall only be interested in type-coherent interpretations, for which $\eta\,\pi \models \Gamma$.

Lemma 3.3: Satisfaction of property assertions is model-independent.
Suppose η_a and η_b are related valuation assignments, type-compatible with the type assignment Γ, and suppose π_a and π_b are related predicate environments, also compatible with Γ. Then

$(\mathcal{R}^\sigma(\mathcal{A}[\![\Gamma \rhd M :: \sigma]\!]\eta_a, \mathcal{B}[\![\Gamma \rhd N :: \sigma]\!]\eta_b) \wedge$
$\mathcal{R}^{\mathit{Pred}\ \sigma}(\mathcal{A}[\![\Gamma \rhd P :: \mathit{Pred}\ \sigma]\!]\pi_a, \mathcal{A}[\![\Gamma \rhd Q :: \mathit{Pred}\ \sigma]\!]\pi_b)) \Rightarrow$
$(\mathcal{A}[\![\Gamma \rhd M :: \sigma ::: P]\!]\eta_a\,\pi_a \Leftrightarrow \mathcal{B}[\![\Gamma \rhd N :: \sigma ::: Q]\!]\eta_b\,\pi_b)$

Proof: The lemma is a direct consequence of the Basic Lemma for models and its extension to predicates, Lemmas 3.1 and 3.2, applied to the meaning of a judgment form.

3.4.5 Parametricity and polymorphic property assertions

There is also a parametricity result for polymorphic property assertions, which is a consequence of Corollary 3.1.

Corollary 3.2: Suppose $\forall\alpha :: \mathit{type} \bullet M ::: P$ is an assertion well-typed under a typing context Γ and closed under a type-compatible value assignment η. Then

$$[\![\Gamma \rhd M ::: P]\!]\eta \equiv \exists\tau :: \mathit{type} \bullet [\![\Gamma[\tau/\alpha] \rhd M]\!]\eta \in [\![\Gamma[\tau/\alpha] \rhd P]\!]$$

Corollary 3:2 can be extended to characterize validity of an assertion in which there occur free term variables not bound by the valuation assignment and in which free predicate variables may occur in the predicate, P.

Definition 3.4: Validity of a polymorphic property assertion.

Suppose an assertion $\forall\alpha :: \mathit{type} \bullet M ::: P$ is well-typed under a typing context Γ but is not necessarily closed under a fixed value assignment η, and that the predicate, P, may contain free predicate variables. The set of predicate variables that occur free in P is denoted by $FV(P)$. Such an assertion is *valid* in a typing context Γ and valuation assignment, η iff

$$\begin{array}{l}\exists\tau :: \mathit{type} \bullet \\ \quad \forall\eta' :: \mathit{FV}(M) \setminus \mathit{Dom}(\eta) \to [\![\,.\,[\![\Gamma \bullet \\ \quad \forall\,\pi :: \mathit{FV}(P) \to \wp([\![\,.\,[\![\Gamma) \bullet \\ \qquad \eta \oplus \eta', \pi \models \Gamma \Rightarrow \\ \qquad\quad [\![\Gamma[\tau/\alpha] \rhd M [\![\eta \oplus \eta' \in [\![\Gamma[\tau/\alpha] \rhd P [\![\pi\end{array}$$

□

Validity of a polymorphically typed assertion can be established at any type instance, but requires the assertion to be satisfied under every valuation and predicate assignment compatible with that type instance. This requirement precludes the validity of an asserted property that is specific to a type substituted for the quantified type variable.

Theorem 3.2: Validity of polymorphic assertions.

If a polymorphic property assertion $\forall\alpha :: \mathit{type} \bullet M ::: P$, well-typed in a typing context, Γ, is valid at any specific type instance, $\Gamma[\tau/\alpha]$, then it is valid at every type instance.

Proof: Immediate from Definition 3.4 and Corollary 3.1.

3.5 Validity of inference rules

A rule asserts a propositional implication of a consequent judgment from zero or more antecedent judgment forms. A rule is *sound* if the propositional implication in terms of which it is formulated is *valid* for a model.

An inference rule typically contains both free term variables and free predicate variables, as a rule formulates how properties of component terms are propagated to composite terms and vice-versa. Furthermore, rules of the core of *P*-logic are polymorphic, as the core logic characterizes just the applicative structures and the free term algebras of Haskell. Theorem 3.1 tells us that a polymorphic property can be observed at any type instance of a quantified type.

Corollary 3.2: The validity of a polymorphic rule of *P*-logic can be observed at any type instance.

4 Checking soundness of rules of *P*-logic

To observe the soundness of a polymorphic rule, we can pick a type instance and for the chosen type, check that the propositional implication of the rule's conclusion from its hypotheses is satisfied under every type-conforming value assignment for term variables that occur in the rule and every type-conforming predicate assignment for its predicate variables. This provides a necessary and sufficient condition to determine that a rule is coherent with the semantics interpretation given by the model. For those rules in which the conclusion asserts a simple property (i.e. a property that does not require a fixed-point), choosing a finite type instance renders soundness as a finite model-checking problem.

For rules that conclude Lfp or Gfp properties, finite model checking alone cannot assure soundness. However, Lemma 3.2 provides the meta-theory on which soundness of a conclusion involving a fixed-point property rests. It gives sufficient conditions for soundness of a Lfp or a Gfp rule, expressed in terms of predicates that are not recursive. The conditions needed to discharge the hypotheses of Lemma 3.2 can indeed be checked with a finite model.

The elements of a model form a type-indexed family of domains, which will guide the choice of specific, finite models for checking soundness of rules. We distinguish properties that can be expressed in terms of the free types of Haskell from those that rely upon an underlying algebra of specific types, or type classes. Rules of *P*-logic express only the properties of free types. For most properties of non-free algebras, *P*-logic relies upon *decision procedures* to determine equivalence of terms modulo a particular algebraic theory, or a combination of such theories. Decision procedures and their compositions must be validated by means external to *P*-logic.

4.1 Finite models for Haskell types

In this section, we consider the type constructions of Haskell, to show how each can be represented by a finite type to model the rules of *P*-logic. The elements of a semantic domain are taken to be either the explicit bottom element, $\perp$, or constants of the type, or pairs of elements, or fixed-length lists of elements.

An element of a datatype is modeled by pairing a constant, which represents the data constructor, with a list of elements representing the arguments of a constructor application. An element of a (finite) function type is modeled by a list of pairs, representing a monotonic function by a trace.

The elements of Bool are $\perp$, *True* and *False*.

The trivial type, *Void*, has two elements, $\perp$ and ().

A product type, (τ_1, τ_2), has the elements $\perp$ and $\{(a, b) \mid a \in \lceil\tau_1\rceil_\perp \text{ and } b \in \lceil\tau_2\rceil_\perp\}$

An arrow type, $\tau_1 \to \tau_2$, has as its elements $\perp$ and the traces of all monotone functions from τ_1 to τ_2.

We choose a representative datatype data *Prelist* $\alpha\beta =$ *Nil* | *Cons* α β. Elements of *Prelist* τ_1 τ_2 are $\perp$, (*Nil*, [ ]) and members of the set $\{(\mathit{Cons}, [a\, b]) \mid a \in \lceil\tau_1\rceil_\perp \text{ and } b \in \lceil\tau_2\rceil_\perp\}$.

Notice that we have not chosen a recursive datatype constructor, such as *List*. It is not necessary to choose a recursively defined type, as no rule of *P*-logic depends upon implicit fixed-points. In fact, the only rules in which there occur terms of datatypes are (9), (10) and (11) in which there are no nested data constructors.

In checking any rule, the principle followed is to choose the least complex type possible to instantiate any type variable of a parametrically polymorphic typed term. Thus, for instance, to check rule (5) for abstraction introduction, we would choose the type $() \to ()$ to model an arrow type.

4.2 Modeling predicates

When a type instance of the term (or terms) in a rule has been chosen, the typing of every predicate in the rule is also determined. To check soundness of the rule, we must simulate all type-compatible predicate assignments to the predicate variables that occur in the rule. The predicate interpretations at a finite type are finite constructions from the predicate constructors that are defined for the type. At every type we have the predicates Univ, $Univ, UnDef and $UnDef. Notice however, that no information is gotten from the assignment of Univ, as this predicate contains every element of the corresponding type domain, or from the assignment of $UnDef, as it is unsatisfied by any domain element.

In addition to the interpretations of $Univ and UnDef, additional predicate interpretations are included in an assignment at a particular type. At a product type, *Pred* (τ_1, τ_2), we add the interpretations of $(P$ :: *Pred* τ_1, Q :: *Pred* $\tau_2)$, where P and Q are fresh predicate variables. For a predicate of an arrow type, $\tau_1 \to \tau_2$, the interpretations of $P \to Q$, where P :: *Pred* τ_1 and Q :: *Pred* τ_2 are added. For datatypes,

Bool adds the interpretations of *True*, *False*, $*True* and $*False*;

Prelist adds the interpretations of *Nil*, *Cons P Q*, $*Nil* and $*Cons P Q*, where P and Q are predicate variables.

4.3 Model checking of inference rules

4.3.1 A machine-interpreted model for Haskell

An abstract semantics for Haskell, coded in Haskell, provides a meta-circular interpreter [4]. This interpreter, while not implementing the IO monad or garbage collection for heap storage, provides an executable model that is simple enough to be certified by inspection as a faithful interpretation of the language definition. With two exceptions, this interpreter meets our need for a model against which to check the soundness of the polymorphic rules of *P*-logic. These exceptions are:

- the bottom element in the semantics domain is interpreted by the Haskell constant undefined, whose evaluation aborts execution of the interpreter;
- function values are interpreted by functions programmed in Haskell, i.e. by explicit abstraction expressions.

A consequence of these design choices in the interpreter is that it aborts execution whenever it calculates a bottom element in the domain—which is unacceptable for modeling predicate satisfaction in *P*-logic. Therefore, the semantics interpreter of [4] has been modified. Domain elements are represented in the following type:

```
data V
  = Void              --- value of type Void
  | Tagged Name [V]   --- data structures
  | TV [V]            --- tuple values
  | FT [(V,V)]        --- trace of a function
  | Bottom            --- the bottom element
```

Function application is modeled using the Haskell library function `lookup` on the trace representation of a finite function.

```
app :: V -> V -> V   --- Application
app (FT trace_elements) x =
   case lookup x trace_elements of
      Nothing -> error ()
      Just r  -> r
```

Using these representation types, it is straightforward to model the domain elements of a finite Haskell type. A domain is calculated by the semantics representation function, *Rep* :: *type* $\to$ V. For instance,

$$
\begin{aligned}
&\mathit{Rep}(\mathit{Prelist\ Void\ Void}) = \\
&\quad \{\mathit{Bottom}, \\
&\quad\ \mathit{Tagged\ ``Nil''}\ \texttt{[]}, \\
&\quad\ \mathit{Tagged\ ``Cons''}\ \texttt{[}\mathit{Bottom,Bottom}\texttt{]}, \\
&\quad\ \mathit{Tagged\ ``Cons''}\ \texttt{[}\mathit{Bottom,Void}\texttt{]}, \\
&\quad\ \mathit{Tagged\ ``Cons''}\ \texttt{[}\mathit{Void,Bottom}\texttt{]}, \\
&\quad\ \mathit{Tagged\ ``Cons''}\ \texttt{[}\mathit{Void,Void}\texttt{]}\}
\end{aligned}
$$

A monotonic, finite function space of type $a \to b$ is calculated algorithmically as the set of all monotonic traces from *Rep a* to *Rep b*.

4.3.2 Automated model checking

An initial step in model-checking a polymorphic rule is the choice of a type instance, justified by Corollary 3.2. Instantiating each universally quantified type variable at the type *Void* meets this requirement. For rules (9) and (10), we choose the type *Prelist Void Void* and for rule (11) we choose a variant of *Prelist* in which the constructor *Cons* is strict in its first argument.

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

Similarly, a predicate assignment binds each predicate variable that occurs free in a rule to a subset of the type domain to which it corresponds. Quantification over predicate assignments is realized by iterating over all possible predicate assignments.

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

For example, the polymorphic rule (6)

$$(\textit{ArrowLeft}) \qquad \frac{\Gamma \vdash N ::: P \qquad M\ N ::: Q \vdash \Delta}{\Gamma,\ M ::: \$(P \to Q) \vdash \Delta}$$

can be checked under the typing assignment N :: *Void*, M :: *Void* $\to$ *Void*, P, Q :: *Pred Void*. For each particular valuation assignment and predicate assignment, we calculate from the antecedent clauses of the rule the weakest context assumption, Γ, and the strongest entailment, Δ, for which all antecedents are true. Then, the truth of the consequent is checked, relative to the calculated context and entailment, using the semantics model to evaluate Haskell terms.

In checking the rule (*ArrowLeft*), the context calculated from the antecedents provides a binding for a term (the variable, N) to which the term variable M in the conclusion can be applied to produce a proposition that logically implies the previously calculated entailment. This succeeds under each valuation and predicate assignment for which the antecedents of the rule could be validated; thus the rule is deemed sound.

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

$$(\textit{Unsound}) \qquad \frac{\Gamma \vdash N ::: P \qquad M\ N ::: Q \vdash \Delta}{\Gamma,\ M ::: (P \to Q) \vdash \Delta}$$

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

5 Conclusions

We have shown that the problem of proving soundness for polymorphically typed inference rules in a modal, mu-calculus can be reduced to finite model-checking. For *P*-logic, which is a new, verification logic for Haskell, soundness is not an obvious property of proposed inference rules, thus an automated technique to check soundness makes an important contribution.

Further development of *P*-logic will include embedding decision procedures for decidable cases of disjoint algebraic theories associated with the class hierarchies of the Haskell type system, such as integer and rational arithmetic, booleans, and an algebra of lists.

Acknowledgements The effort to prove *P*-logic sound was encourged by the entire Programatica team, but especially by Jim Hook. The author is also indebted to Bill Harrison for critical reading and comments on earlier drafts.

References

[1] M. Abadi, B. Pierce, and G. Plotkin. Faithful ideal models for recursive polymorphic types. In *Proceedings of Fourth Annual Symposium on Logic in Computer Science*, pages 216–225. IEEE Computer Society Press, June 1989.

[2] P. J. de Bruin. *Inductive types in programming languages*. PhD thesis, University of Groningen, 1995.

[3] M. M. Fokkinga. *Law and Order in Algorithmics*. PhD thesis, University of Twente, Twente, The Netherlands, Feb. 1992.

[4] W. Harrison, T. Sheard, and J. Hook. Fine control of demand in Haskell. In *Sixth International Conference on the Mathematics of Program Construction*, volume 2386 of *Lecture Notes in Computer Science*, pages 68–93. Springer Verlag, July 2002.

[5] F. Honsell and D. Sannella. Pre-logical relations. In *Computer Science Logic, CSL'99*, volume 1683 of *Lecture Notes in Computer Science*, pages 546–561. Springer Verlag, 1999.

[6] R. B. Kieburtz. P-logic: Property verification for Haskell programs. 2002.

[7] D. MacQueen, G. Plotkin, and R. Sethi. An ideal model for recursive polymorphic types. *Information and Control*, 71(1/2):95–130, Oct. 1986.

[8] J. C. Mitchell. *Foundations for Programming Languages*. MIT Press, 1996.

[9] J. C. Mitchell and A. R. Meyer. Second-order logical relations. In *Logics of Programs*, volume 193 of *Lecture Notes in Computer Science*, pages 225–236. Springer Verlag, June 1985.

[10] J. C. Reynolds. Types, abstraction and parametric polymorphism. In *Information Processing 83*, pages 513–523. North-Holland, Amsterdam, 1983.

[11] P. Wadler. Theorems for free! In *Proc. of 4th ACM Conf. on Functional Programming Languages and Computer Architecture*, pages 347–359. ACM Press, Sept. 1989.

Appendix: Proofs omitted in the text :

Definition A.1: Logical relations over pointed c.p.o. domains.
Let $\mathcal{A}$ and $\mathcal{B}$ be $\lambda^{\rightarrow,\times,fix}$ domains for a signature, Σ, that includes polymorphic operators $\{\mathbf{Proj}_1, \mathbf{Proj}_2, \mathbf{App}, \mathbf{Fix}\}$ and a type-indexed family of constants, Σ^σ.

A *logical relation* over full continuous hierarchies $\mathcal{A}$ and $\mathcal{B}$ is a family of relations indexed by type expressions over a signature Σ such that:

1. $R^\sigma \subseteq A^\sigma \times B^\sigma$ for each type σ
2. $R^\sigma(\mathit{Const}_\mathcal{A}(c), \mathit{Const}_\mathcal{B}(c))$ for every typed constant $c \in \Sigma^\sigma$.
3. $R^{\sigma\rightarrow\tau}(f,g)$ iff $\forall x \in A^\sigma,\ y \in B^\sigma \bullet R^\sigma(x,y) \Rightarrow R^\tau(\mathbf{App}_\mathcal{A}\ f\ x,\ \mathbf{App}_\mathcal{B}\ g\ y)$
4. $R^{\sigma\times\tau}(p,q)$ iff $R^\sigma(\mathbf{Proj}_1\ p,\ \mathbf{Proj}_1\ q)$ and $R^\tau(\mathbf{Proj}_2\ p,\ \mathbf{Proj}_2\ q)$
5. $\mathcal{R}$ is directed-complete as a unary predicate over $\mathcal{A}\times\mathcal{B}$; i.e. limits of related directed sets are related
6. For all $a' \sqsubseteq a$ in A^σ and $b' \sqsubseteq b$ in B^σ, $R^\sigma(a,b) \Rightarrow R^\sigma(a',b')$
7. $R^{\sigma\rightarrow\sigma}(f,g) \Rightarrow R^\sigma(\mathbf{Fix}_\mathcal{A}\ f,\ \mathbf{Fix}_\mathcal{B}\ g)$

□

This definition is essentially that given by Mitchell [8], and extends the basic definition by addition of the fifth and sixth clauses, which require a relation to be directed-complete and downward-closed with respect to the order structures of related domains, and the seventh clause, which extends the relation to fixpoints.

Lemma 3.1: Let $\sigma,\ \sigma_1,\ldots,\sigma_n,\ \tau$ be pointed types and suppose $\mathcal{R} \subseteq \mathcal{A}\times\mathcal{B}$ is logical. Then

- $\mathcal{R}^{\mathit{Pred}\ \sigma}(\mathcal{A}[\![\mathsf{Univ}]\!], \mathcal{B}[\![\mathsf{Univ}]\!])$ and $\mathcal{R}^{\mathit{Pred}\ \sigma}(\mathcal{A}[\![\mathsf{UnDef}]\!], \mathcal{B}[\![\mathsf{UnDef}]\!])$ at each type, σ;
- $\mathcal{R}^{\mathit{Pred}\ \sigma}(\mathcal{A}[\![\$P]\!], \mathcal{B}[\![\$Q]\!]) \Leftrightarrow \mathcal{R}^{\mathit{Pred}\ \sigma}(\mathcal{A}[\![P]\!], \mathcal{B}[\![Q]\!])$
- $\mathcal{R}^{\mathit{Pred}\ \sigma}(\mathcal{A}[\![P_1]\!], \mathcal{B}[\![P_2]\!]) \wedge \mathcal{R}^{\mathit{Pred}\ (\sigma\rightarrow\tau)}(\mathcal{A}[\![P_1 \rightarrow Q_1]\!],\ \mathcal{B}[\![P_2 \rightarrow Q_2]\!]) \Rightarrow \mathcal{R}^{\mathit{Pred}\ \tau}(\mathcal{A}[\![Q_1]\!], \mathcal{B}[\![Q_2]\!])$
- $\mathcal{R}^{\mathit{Pred}\ \sigma}(\mathcal{A}[\![C^n P_1 \ldots P_n]\!], \mathcal{B}[\![C^n Q_1 \ldots Q_n]\!]) \Leftrightarrow \forall i \in 1\,..\,n \bullet \mathcal{R}^{\mathit{Pred}\ \sigma_i}(\mathcal{A}[\![\overline{P_i}]\!],\ \mathcal{B}[\![\overline{Q_i}]\!])$
  where $C^n :: \sigma_1 \times \cdots \times \sigma_n \rightarrow \sigma$
  and $\overline{P_i} = \begin{cases} \$P_i & \text{if } C^{(n)} \text{ is strict in its } i^{th} \text{ argument} \\ P_i & \text{otherwise} \end{cases}$
  and $\overline{Q_i}$ is defined similarly
- $\mathcal{R}^{\mathit{Pred}\ \sigma}(\mathcal{A}[\![P_1]\!], \mathcal{B}[\![P_2]\!]) \wedge \mathcal{R}^{\mathit{Pred}\ \sigma}(\mathcal{A}[\![Q_1]\!],\ \mathcal{B}[\![Q_2]\!]) \Rightarrow \mathcal{R}^{\mathit{Pred}\ \sigma}(\mathcal{A}[\![P_1 \wedge Q_1]\!], \mathcal{B}[\![P_2 \wedge Q_2]\!])$
- $\mathcal{R}^{\mathit{Pred}\ \sigma}(\mathcal{A}[\![P_1]\!], \mathcal{B}[\![P_2]\!]) \vee \mathcal{R}^{\mathit{Pred}\ \sigma}(\mathcal{A}[\![Q_1]\!],\ \mathcal{B}[\![Q_2]\!]) \Rightarrow \mathcal{R}^{\mathit{Pred}\ \sigma}(\mathcal{A}[\![P_1 \vee Q_1]\!], \mathcal{B}[\![P_2 \vee Q_2]\!])$

Proof: The proof is by induction on the structure of non-recursive predicates. For each form of predicate, we show that the semantics of a ground term at the given type satisfies the relation iff it belongs to the set specified by the predicate in both the models $\mathcal{A}$ and $\mathcal{B}$.

case Univ :: *Pred* σ: Since $\mathcal{A}[\![\mathsf{Univ}]\!]^{\mathit{Pred}\ \sigma} = \mathcal{A}\lceil\sigma\rceil_\perp$ and $\mathcal{B}[\![\mathsf{Univ}]\!]^{\mathit{Pred}\ \sigma} = \mathcal{B}\lceil\sigma\rceil_\perp$, conclude that $\forall t \bullet \mathcal{A}[\![t]\!]^\sigma \in \mathcal{A}[\![\mathsf{Univ}]\!]^{\mathit{Pred}\ \sigma}$ and $\mathcal{B}[\![t]\!]^\sigma \in \mathcal{B}[\![\mathsf{Univ}]\!]^{\mathit{Pred}\ \sigma}$

case UnDef :: *Pred* σ:
$\forall t :: \sigma \bullet \mathcal{A}[\![t]\!] = \perp_\mathcal{A} \in \mathcal{A}[\![\mathsf{UnDef}]\!] \Rightarrow$
$\exists t' :: \sigma \bullet \mathcal{B}[\![t']\!] = \perp_\mathcal{B} \in \mathcal{B}[\![\mathsf{UnDef}]\!]$ and $\mathcal{R}^\sigma(\perp_\mathcal{A}, \perp_\mathcal{B})$

case $\$P$ :: *Pred* σ: Since $\mathcal{R}$ is downward-closed, $\mathcal{R}^{\mathit{Pred}\ \sigma}(\mathcal{A}[\![P]\!], \mathcal{B}[\![P]\!]) \Leftrightarrow \mathcal{R}^{\mathit{Pred}\ \sigma}(\mathcal{A}[\![P]\!] \setminus \{\perp\}, \mathcal{B}[\![P]\!] \setminus \{\perp\})$

case $P \rightarrow Q$: $\forall f \in \mathcal{A}[\![P_1 \rightarrow Q_1]\!], g \in \mathcal{B}[\![P_2 \rightarrow Q_2]\!] \bullet \mathcal{R}^{\sigma\rightarrow\tau}(f,g)$ There are four cases:

- $f = \perp_\mathcal{A}$ and $g = \perp_\mathcal{B}$ Then $\forall a \in \mathcal{A}[\![P_1]\!],\ b \in \mathcal{B}[\![P_2]\!] \bullet \mathbf{App}_\mathcal{A}\ f\ a = \perp_\mathcal{A}$ and $\mathbf{App}_\mathcal{B}\ g\ b = \perp_\mathcal{B}$ and $\mathcal{R}^\tau(\perp_\mathcal{A}, \perp_\mathcal{B})$
- $f = \perp_\mathcal{A}$ and $g \neq \perp_\mathcal{B}$ but $\forall b \in \mathcal{B}[\![P_2]\!] \bullet \mathbf{App}_\mathcal{B}\ g\ b = \perp_\mathcal{B}$ and $\mathcal{R}^\tau(\perp_\mathcal{A}, \perp_\mathcal{B})$
- $f \neq \perp_\mathcal{A}$ and $g = \perp_\mathcal{B}$ but $\forall a \in \mathcal{A}[\![P_1]\!] \bullet \mathbf{App}_\mathcal{A}\ f\ a = \perp_\mathcal{A}$ and $\mathcal{R}^\tau(\perp_\mathcal{A}, \perp_\mathcal{B})$
- $f \neq \perp_\mathcal{A}$ and $g \neq \perp_\mathcal{B}$ and $\forall a \in \mathcal{A}[\![P_1]\!],\ b \in \mathcal{B}[\![P_2]\!] \bullet \mathcal{R}^\tau(\mathbf{App}_\mathcal{A}\ f\ a,\ \mathbf{App}_\mathcal{B}\ g\ b)$ with $\mathbf{App}_\mathcal{A}\ f\ a \in \mathcal{A}[\![Q_1]\!]$ and $\mathbf{App}_\mathcal{B}\ g\ b \in \mathcal{B}[\![Q_2]\!]$

$\Leftrightarrow\ \mathcal{R}^{\mathit{Pred}\ (\sigma\rightarrow\tau)}(\mathcal{A}[\![P_1 \rightarrow Q_1]\!],\ \mathcal{B}[\![P_2 \rightarrow Q_2]\!])$

case $C^{(0)}$ where $\sigma = \mathbf{data}\ C^{(0)} \mid \cdots \mid C^{(n)}\tau_1 \cdots \tau_n$:
$\forall a \in \mathcal{A}\lceil\sigma\rceil \bullet a \in \mathcal{A}[\![C^{(0)}]\!]$ iff $a = C^{(0)}_\mathcal{A}$ and
$\forall b \in \mathcal{B}\lceil\sigma\rceil \bullet b \in \mathcal{B}[\![C^{(0)}]\!]$ iff $b = C^{(0)}_\mathcal{B}$ and
$\mathcal{R}^\sigma(C^{(0)}_\mathcal{A}, C^{(0)}_\mathcal{B})$
$\Leftrightarrow\ \mathcal{R}^{\mathit{Pred}\ \sigma}(\mathcal{A}[\![C^{(0)}]\!], \mathcal{B}[\![C^{(0)}]\!])$

case $C^{(n)}[P_1 \cdots P_n]$ where $\sigma = \mathbf{data}\ C^{(0)} \mid \cdots \mid C^{(n)}\tau_1 \cdots \tau_n$:
$\mathcal{R}^{\mathit{Pred}\ \tau_1}(\mathcal{A}[\![P_1]\!], \mathcal{B}[\![Q_1]\!]) \cdots \mathcal{R}^{\mathit{Pred}\ \tau_n}(\mathcal{A}[\![P_n]\!], \mathcal{B}[\![Q_n]\!]) \Rightarrow$
$(\forall a_1 \in \mathcal{A}[\![P_1]\!] \cdots a_n \in \mathcal{A}[\![P_n]\!] \bullet$
$\forall b_1 \in \mathcal{B}[\![Q_1]\!] \cdots b_n \in \mathcal{B}[\![Q_n]\!] \bullet$
$\mathcal{R}^{\tau_1}(a_1,b_1) \wedge \cdots \wedge \mathcal{R}^{\tau_n}(a_n,b_n))$ and
$\forall a \in \mathcal{A}\lceil\sigma\rceil_\perp \bullet a \in \mathcal{A}[\![C^{(n)}P_1 \cdots P_n]\!]$ iff
$\exists a_1 \in \mathcal{A}[\![P_1]\!] \cdots a_n \in \mathcal{A}[\![P_n]\!] \bullet a = C^{(n)}_\mathcal{A} a_1 \ldots a_n$ and
$\forall b \in \mathcal{B}\lceil\sigma\rceil_\perp \bullet b \in \mathcal{B}[\![C^{(n)}P_1 \cdots P_n]\!]$ iff
$\exists b_1 \in \mathcal{B}[\![P_1]\!] \cdots b_n \in \mathcal{B}[\![P_n]\!] \bullet b = C^{(n)}_\mathcal{B} b_1 \ldots b_n$ and
$\mathcal{R}^{\tau_1}(a_1,b_1) \wedge \cdots \wedge \mathcal{R}^{\tau_n}(a_n,b_n)) \Rightarrow$
$\mathcal{R}^\sigma(\mathcal{A}[\![C^{(n)}a_1 \ldots a_n]\!], \mathcal{B}[\![C^{(0)}b_1 \ldots b_n]\!])$
$\Leftrightarrow\ \mathcal{R}^{\mathit{Pred}\ \sigma}(\mathcal{A}[\![C^{(n)}P_1 \cdots P_n]\!],\ \mathcal{B}[\![C^{(n)}Q_1 \cdots Q_n]\!])$

(The argument is similar when the constructor $C^{(n)}$ is strict in one or more arguments, except that the predicates in strict argument positions are strengthened.)

case $P \wedge Q$: $\mathcal{R}^{Pred\ \sigma}(\mathcal{A}\llbracket P\rrbracket, \mathcal{B}\llbracket P\rrbracket)$ and
$\mathcal{R}^{Pred\ \sigma}(\mathcal{A}\llbracket Q\rrbracket, \mathcal{B}\llbracket Q\rrbracket) \Rightarrow \mathcal{R}^{Pred\ \sigma}(\mathcal{A}\llbracket P\rrbracket \cap \mathcal{A}\llbracket Q\rrbracket, \mathcal{B}\llbracket P\rrbracket \cap \mathcal{B}\llbracket Q\rrbracket)$

case $P \vee Q$: $\mathcal{R}^{Pred\ \sigma}(\mathcal{A}\llbracket P\rrbracket, \mathcal{B}\llbracket P\rrbracket)$ and
$\mathcal{R}^{Pred\ \sigma}(\mathcal{A}\llbracket Q\rrbracket, \mathcal{B}\llbracket Q\rrbracket) \Rightarrow \mathcal{R}^{Pred\ \sigma}(\mathcal{A}\llbracket P\rrbracket \cup \mathcal{A}\llbracket Q\rrbracket, \mathcal{B}\llbracket P\rrbracket \cup \mathcal{B}\llbracket Q\rrbracket)$

□

Lemma 3.2 Let σ be a pointed type and suppose $\mathcal{R} \subseteq \mathcal{A}\times\mathcal{B}$ is logical. Then

$\mathcal{R}^{Pred\ \sigma}(\mathcal{A}\llbracket \mathsf{Lfp}\,\xi \bullet H\rrbracket,\ \mathcal{B}\llbracket \mathsf{Lfp}\,\xi \bullet H\rrbracket)$ and
$\mathcal{R}^{Pred\ \sigma}(\mathcal{A}\llbracket \mathsf{Gfp}\,\xi \bullet H\rrbracket,\ \mathcal{B}\llbracket \mathsf{Lfp}\,\xi \bullet H\rrbracket)$
iff $\exists P :: \mathit{Pred}\ \sigma \bullet \mathcal{R}^{Pred\ \sigma}(\mathcal{A}\llbracket P\rrbracket,\ \mathcal{B}\llbracket P\rrbracket) \Rightarrow$
$\mathcal{R}^{Pred\ \sigma}(\mathcal{A}\llbracket H[P/\xi]\rrbracket,\ \mathcal{B}\llbracket H[P/\xi]\rrbracket)$

Proof: For the "if" direction of the bi-implication, we can use natural induction for each of the l.f.p. and g.f.p. forms. Suppose the right-hand side of the bi-implication holds. Referring to the semantics of the l.f.p. formula, we shall show that every finite union relates to itself.

For an l.f.p. predicate, the inductive hypothesis is

$$\mathcal{R}^{\sigma}(\bigcup_{i=0}^{n} \mathcal{A}\llbracket H^i[\mathsf{UnDef}]\rrbracket,\ \bigcup_{i=0}^{n} \mathcal{B}\llbracket H^i[\mathsf{UnDef}]\rrbracket)$$

The base case for the induction is furnished by

$$\mathcal{R}^{\sigma}(\bot_{\mathcal{A}}, \bot_{\mathcal{B}}) \Leftrightarrow \mathcal{R}^{Pred\ \sigma}(\mathcal{A}\llbracket H^0[\mathsf{UnDef}]\rrbracket, \mathcal{B}\llbracket H^0[\mathsf{UnDef}]\rrbracket)$$

since σ is pointed and $\llbracket \mathsf{UnDef}\rrbracket^{\sigma} = \{\bot\}$ in every model.

For the induction step, substitute $H^n[\mathsf{UnDef}]$ for the existentially bound predicate variable P in the right-hand side of the bi-implication and conclude from the inductive hypothesis and the definition of $H^i[P]$

$$\mathcal{R}^{Pred\ \sigma}(\bigcup_{i=0}^{n+1} \mathcal{A}\llbracket H^i[\mathsf{UnDef}]\rrbracket,\ \bigcup_{i=0}^{n+1} \mathcal{B}\llbracket H^i[\mathsf{UnDef}]\rrbracket)$$

Thus, by natural induction, the semantics of n-times iterations of H in $\mathcal{A}$ and $\mathcal{B}$ are related at each natural number, n. The nested unions are directed sets. We invoke the property that a logical relation is closed under limits of finite directed sets to conclude that the semantics of the l.f.p. formula are related.

The "only-if" direction of the bi-implication follows immediately from the property of a fixed-point, $\mathsf{Lfp}\xi \bullet H = H[\mathsf{Lfp}\xi \bullet H/\xi]$.

For g.f.p. formulas, a similar inductive argument, using for its base case $\mathcal{R}^{Pred\ \sigma}(\mathcal{A}\llbracket \mathsf{Univ}\rrbracket, \mathcal{B}\llbracket \mathsf{Univ}\rrbracket)$, shows that the relation includes all finite intersections from the definition of g.f.p. semantics. Downward closure of R assures the existence of a limit, hence the $\mathcal{A}$ and $\mathcal{B}$ semantics of the g.f.p. are also related.

Theorem 3.1: Parametricity for predicates.

Let $\mathcal{R} \subseteq \mathcal{A} \times \mathcal{B}$ be a logical relation. Then the following are logically equivalent conditions for the induced relation of predicates P and Q at a generalized type:

$\mathcal{R}^{\forall\alpha.Pred\ \sigma}(\mathcal{A}\llbracket P\rrbracket, \mathcal{B}\llbracket Q\rrbracket)$ and
$\exists\tau :: \mathit{type} \bullet \mathcal{R}^{Pred\ \sigma[\tau/\alpha]}(\mathcal{A}\llbracket P\rrbracket, \mathcal{B}\llbracket Q\rrbracket)$

Proof: From Corollary 3.1 and the definition of logical relations extended to predicates we have

$$\begin{array}{ll} & \mathcal{R}^{\forall\alpha.Pred\ \sigma}(\mathcal{A}\llbracket P\rrbracket, \mathcal{B}\llbracket Q\rrbracket) = \\ & \quad \bigcup_{\tau::\mathit{type}} \mathcal{R}^{Pred\ \sigma[\tau/\alpha]}(\mathcal{A}\llbracket P\rrbracket, \mathcal{B}\llbracket Q\rrbracket) \\ \equiv & \exists\tau :: \mathit{type} \bullet \forall a, b \bullet a \in \mathcal{A}\llbracket P\rrbracket^{\sigma[\tau/\alpha]} \wedge \\ & \qquad b \in \mathcal{B}\llbracket Q\rrbracket^{\sigma[\tau/\alpha]} \Leftrightarrow \mathcal{R}^{\sigma[\tau/\alpha]}(a, b) \\ \equiv & \exists\tau :: \mathit{type} \bullet \mathcal{R}^{Pred\ \sigma[\tau/\alpha]}(\mathcal{A}\llbracket P\rrbracket, \mathcal{B}\llbracket Q\rrbracket) \end{array}$$

□