A Logic for Haskell

Dick Kieburtz
OGI School of Science & Engineering
Oregon Health & Science University
September 25, 2001
Updated April 17, 2003
Overall Objectives

• A verification logic for Programatica
  – To support formal reasoning about properties of programs
  – The term language is Haskell 98
    • Initially, omitting monads and classes
  – First-order predicate formulas with equality
    • Extended to a modal $\mu$-calculus
      – $\mu$-calculus adds least and greatest fixed-point formulas
      – A modality designates predicates that require evaluation of a term for their satisfaction

• A tool to certify properties asserted of a program by (interactive) proof construction
  – Libraries of proof strategies suggested by human intuition will be programmed to use in certifying properties by verification
Technical Approach

• **Define a logic whose standard interpretation is given in terms of Haskell semantics**
  – Programatica logic expresses properties of well-typed Haskell terms
    • Avoids translating to a more primitive modeling language

• **Check soundness of each rule of the logic with respect to a Haskell semantics model**
  – Semantics is formulated independently of the logic

• **Develop strategies for computational proof construction**
  – To support verification of program properties with machine-checked proofs
This Talk

• Introduction to the Programatica logic
• Semantic interpretation of the logic
• Inference rules
• Soundness
• Overview of tool support
P-logic

- A modal logic for Haskell
  - **Predicates** range over Haskell terms
  - **Predicate formulas** are constructed with
    - lifted data constructors (term congruence operators)
    - propositional connectives
    - least and greatest fixed-point binders, Lfp and Gfp
    - $\$-modality designates a well-definedness requirement
  - **Congruence formulas** relate properties to the shapes of terms
    - e.g. the formula \( (P : Q) \), where \( P \) and \( Q \) are formulas, is satisfied by a Haskell term \( (h : t) \) where \( h \) satisfies \( P \) and \( t \) satisfies \( Q \)
  - **Lfp and Gfp formulas** assert universal/existental properties of (unbounded) term structures

- Congruence formulas relate properties to the shapes of terms
  - e.g. the formula \( (P : Q) \), where \( P \) and \( Q \) are formulas, is satisfied by a Haskell term \( (h : t) \) where \( h \) satisfies \( P \) and \( t \) satisfies \( Q \)
  - **Lfp and Gfp formulas** assert universal/existental properties of (unbounded) term structures
A Syntax of Formulas

Propositions:

\[ M :: P \quad -- \text{asserts that } M \text{ satisfies } P \]
\[ M \equiv N \quad -- \text{asserts (semantic) equality of } M \text{ and } N \]

Unary Predicates:

\[ P ::= \text{Univ} \quad -- \text{the universal predicate} \]
\[ \mid \text{Undef} \quad -- \text{the predicate satisfied only by } \bot \]
\[ \mid P_1 \land P_2 \quad -- \text{a conjunctive predicate formula} \]
\[ \mid P_1 \lor P_2 \quad -- \text{a disjunctive predicate formula} \]
\[ \mid P_1 \rightarrow P_2 \quad -- \text{an “arrow” formula} \]
\[ \mid \$P \quad -- \text{a strong predicate (requires well-definedness)} \]
\[ \mid \text{Lfp } \xi \cdot P \quad -- \text{a least fixpoint (LFP) formula} \]
\[ \mid \text{Gfp } \xi \cdot P \quad -- \text{a greatest fixpoint (GFP) formula} \]
\[ \mid C \ P_1 \ldots P_k \quad -- \text{a term congruence formula} \]
\[ \quad \bullet \text{ Where } C \text{ is a “lifted” data constructor of arity } k \]
\[ \mid !(\llcorner C) \quad -- \text{“lifted” sections} \]
\[ \mid \{ | \text{pattern} | \text{Prop} \} \quad -- \text{a set comprehension} \]
Expressing Properties of Terms

• How can we express the property (of a list-typed value) of finiteness?
  – In first-order logic, it’s not possible to express the condition that a list is finite, without resorting to recursion
  – In a higher-order logic, inductive formulas are available
    • Induction rules quantify over predicates, e.g.
      \[ \text{Finite-list}(A) = [P] P \rightarrow (A \rightarrow P \rightarrow P) \rightarrow P \]
      This formula gives a type of finite lists, but does not directly describe their structure as Haskell terms

• A better solution
  – Introduce recursion in predicate definitions
    • mu-calculus (Kozen, 1983)
  – Lift the constructors of terms to the status of predicates (analogous to pattern constructors)
Term Congruence Formulas

• Taking advantage of the isomorphism between a free datatype and the sum-of-products of its component types
  – Haskell exploits the isomorphism in pattern-matches
  – Programatica logic exploits the isomorphism with congruence formulas

The proposition $M :::: C P_1 \ldots P_k$ is equivalent to:

$\exists N_1 \ldots N_k \cdot M = C N_1 \ldots N_k \land (N_1 :::: P_1) \land \ldots \land (N_k :::: P_k)$

• Congruence formulas are succinct, and
  – Coherent with the interpretation of $P$-logic (to follow)
  – The isomorphism between structure and components leads directly to inference rules for congruence formulas (to follow)
A logic for reasoning about partial, continuous functions

• Strong and weak assertions
  – A strong assertion, $M :: \ P$, is satisfied if term $M$ has a defined value which satisfies $P$
  – A weak assertion, $N :: Q$, is satisfied if term $N$ is undefined or has a defined value which satisfies $Q$

• Assertions about a function $f :: \tau_1 \to \tau_2$
  $f :: ¥(¥Univ \to ¥Univ)$ asserts that $f$ is total
  $f :: \text{UnDef} \to \text{UnDef}$ asserts that $f$ is strict
  $f :: \text{Univ} \to ¥\text{Univ}$ asserts that $f$ is a constant fn
  (since $f$ is presumed continuous)
Predicate Formulas

• Propositional connectives are lifted to connectives of unary predicate formulas

\[
\begin{align*}
  x : & : (P \land Q) \equiv_{\text{def}} x : : P \land x : : Q \\
  x : & : (P \lor Q) \equiv_{\text{def}} x : : P \lor x : : Q \\
  x : & : \$(P \land Q) \equiv_{\text{def}} x : : \$P \land x : : \$Q \\
  x : & : \$(P \lor Q) \equiv_{\text{def}} x : : \$P \lor x : : \$Q \\
  x : & : \neg P \equiv_{\text{def}} \neg (x : : P) \lor x : : \text{UnDef}
\end{align*}
\]
Recursively Defined Predicates

- $\mu$-calculus extends first-order logic with least and greatest fixed-point formulas
  - Expresses properties asserted over the extent of a data structure

Examples:
- $\text{Finite-list} = \text{Lfp } \zeta \bullet [ ] \lor (\text{Univ} : \$\zeta)$
- $\text{Head-strict-list} = \text{Lfp } \zeta \bullet [ ] \lor (\$\text{Univ} : \zeta)$
- $\text{Infinite-Stream} = \text{Gfp } \zeta \bullet (\text{Univ} : \$\zeta)$
Example 1: length of lists is additive

- Functions as defined in Haskell

  ```haskell
  length :: [a] -> Integer
  length [] = 0
  length (_:t) = 1 + length t
  ```

  - the multi-equation function definition is desugared to yield a single equation:

  ```haskell
  length = \_xs -> case _xs of
         [] -> 0
         (_:t) -> 1 + length t
  ```

  - Similarly,

  ```haskell
  (++) = \_xs ys -> case _xs of
         [] -> ys
         (x:xs) -> x : (++) xs ys
  ```

  - We assert the following

    ```haskell
    assert All xs, ys :::: Finite-list :: length (xs ++ ys) $== length xs + length ys
    ```

    Proved using an inductive proof rule for list equality
Example 2: Correctness of a factorial function

- Functions as defined in Haskell
  - A generalized primitive recursion combinator:
    \[ \text{genPR} :: (a \to \text{Bool}) \to (a \to a) \to c \to (a \to c \to c) \to a \to c \]
    \[ \text{genPR} \ p \ b \ g \ h \ x = \begin{cases} \text{if} \ p \ x \ \text{then} \ g \ \text{else} \ h \ x \ (\text{genPR} \ p \ b \ g \ h \ (b \ x)) \end{cases} \]
  - A factorial function:
    \[ \text{fact} :: \text{Integer} \to \text{Integer} \]
    \[ \text{fact} = \text{genPR} \ \text{eq0} \ (\text{subtract} \ 1) \ 1 \ (*) \]
  - We assert the following
    property \( x \geq 0 \Rightarrow \text{fact} \ x = x! \)
    where \( 0! = 1 \)
    \( (x+1)! = (x+1) \times x! \)
    Proved using an inductive proof rule for genPR
    This rule requires that a set well-ordered by \( p \) and \( b \) be specified:
    \[ \text{WO} \ p \ b = \text{Lfp} \ \eta \cdot \{ \ | x \ | \ p \ x \ \Downarrow \text{True} \lor (b \ x :::: \Downarrow \eta) \} \]
Example 3: Ordered insertion in a list

- **Functions as defined in Haskell**
  
  ```haskell
  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
  ```

  - the multi-equation function definition is desugared to yield a single equation:
    
    ```haskell
    insert a = \_ys \rightarrow case _ys of
      [] \rightarrow [a]
      (y : _) | a < y \rightarrow a : ys
               | a == y \rightarrow ys
      (y : ys) \rightarrow y : insert a ys
    ```

  - We assert the following
    
    ```haskell
    assert All xs • xs ::: \$Univ ⇒ insert a xs ::: !(≪a) unless !(=⇒a)
    ```

    where
    
    ```haskell
    P unless Q = Gfp ζ • (︦Q : Univ) ∨ (P : ︦ζ)
    ```

    and
    
    ```haskell
    □P = Gfp ζ • [] ∨ (P : ︦ζ)
    ```

  This property has been proved using the Gfp rule (and many others)
Semantic Interpretation
Semantic Interpretation of Formulas

• Predicate formulas are interpreted as characteristic predicates of sets (posets) in a semantics domain for Haskell
  – A formula is interpreted in a type (or type scheme)

• Notation:
  \( \lceil \tau \rceil \) is the set of domain elements of the Haskell type \( \tau \) (an ideal\(^*\))
  \( C_{[\tau]} \) is the interpretation of the constant symbol \( C \) in the type \( \tau \)
  \( \llbracket P \rrbracket^\tau \) is the ideal of domain elements \( \{ t \in \lceil \tau \rceil \mid t \text{ satisfies } P \} \), where \( P \) is an unstrengthened predicate
  \( \llbracket \$P \rrbracket^\tau \) is the set of elements \( \llbracket P \rrbracket^\tau - \{ \bot \} \)

\(^*\) (Recall that an ideal poset is downward-closed and contains limits of its finite directed subsets.)
For Example:
Distinguished Predicates

**Strong modality**

\[
\llbracket \text{Univ} \rrbracket^\tau = \llbracket \tau \rrbracket - \{\bot\}
\]

\[
\llbracket \text{UnDef} \rrbracket^\tau = \{\}\]

**Weak modality**

\[
\llbracket \text{UnDef} \rrbracket^\tau = \{\bot\}
\]
Predicates derived from Sections

• Equality comparisons with constants
  \[ !!(==a) \]^{\tau} = \{ x \in \llbracket \tau \rrbracket \mid x = a_{\llbracket \tau \rrbracket} \lor x = \bot \}\]

• Ordering relations
  \[ !!(<a) \]^{\tau} = \{ x \in \llbracket \tau \rrbracket \mid x <_{\llbracket \tau \rrbracket} a_{\llbracket \tau \rrbracket} \lor x = \bot \}\]
  where \( \tau \) is an instance of the Ord type class.
Term Congruence Predicates

- A datatype definition populates a signature $\Sigma_k^\tau$ with its data constructors of arity $k$ (for $k \geq 0$)

$$(C, (\tau_1, \ldots, \tau_k)) \in \Sigma_k^\tau \Rightarrow$$

$$\llbracket C \ P_1 \ldots \ P_k \rrbracket^\tau =$$

$$\{C^{\!\tau} \ x_1 \ldots x_k \mid x_1 \in \llbracket P_1 \rrbracket^{\tau_1} \land \ldots \land x_k \in \llbracket P_k \rrbracket^{\tau_k}\} \cup \{\bot\}$$

Arrow Predicates

$$\llbracket P \rightarrow Q \rrbracket^{\tau_1 \rightarrow \tau_2} =$$

$$\{f \in \llbracket \tau_1 \rrbracket \rightarrow \llbracket \tau_2 \rrbracket \mid \forall x \in \llbracket P \rrbracket^{\tau_1} \bullet f \ x \in \llbracket Q \rrbracket^{\tau_2}\} \cup \{\bot\}$$
Conjunction and Disjunction

\[ [P_1 \land P_2]^\tau = [P_1]^\tau_1 \cap [P_2]^\tau \]
\[ [P_1 \lor P_2]^\tau = [P_1]^\tau_1 \cup [P_2]^\tau \]

The Equality Predicate

\[ [(\equiv)]^\tau = \{(u, v) \mid u \in \tau \land v \in \tau \land u = v \} \]
\[ [(\not\equiv)]^\tau = \{(u, v) \mid u \in \tau \land v \in \tau \land u = v \land u \neq \bot \} \]
**Fixed-Point Formulas**

*H* is a predicate formula, *admissible* for fixed-point binding of the predicate variable *ζ* if *ζ* does not occur in a negated position.

**LFP:**
\[
\llbracket \text{Lfp} \xi \bullet H \rrbracket^\tau = \bigcup_{j=0}^{\infty} \llbracket H^j \rrbracket^\tau
\]
where \( H^0 = \text{UnDef} \)
\[
H^{j+1} = H \ [H^j / \xi]
\]

**GFP:**
\[
\llbracket \text{Gfp} \xi \bullet H \rrbracket^\tau = \bigcap_{j=0}^{\infty} \llbracket H^j \rrbracket^\tau
\]
where \( H^0 = \text{Univ} \)
\[
H^{j+1} = H \ [H^j / \xi]
\]
Example 1: Tail-strict lists

- Consider LFP formula
  \[ \text{Strict-list}(A) \equiv \text{Lfp } \xi \cdot [ ] \lor (\text{\$A : \$} \xi) \]
  where \textit{data unit}  \text{\$A } = A

- The interpretation of \text{Strict-list}(A) is
  \[
  \{\bot\} \cup \{\bot, [ ]\} \cup \{\bot, [ ], [A]\} \cup \{\bot, [ ], [A], [A, A]\} \cup ...
  \]

This is the representation of a flat subdomain (the \bot element is never embedded in a list structure)
Example 2: Non-tail-strict lists

- Consider LFP formula

\[ \text{Non-strict}(A) \equiv \text{Lfp } \xi \cdot [\ ] \lor (A : \xi) \]

where \( \text{data unit}A = A \)

- The interpretation of \( \text{Non-strict}(A) \) is

\[
\{ \bot \} \cup \{ \bot, [\ ], (\bot: \bot), (A: \bot) \} \cup \{ \bot, [\ ], [\bot], [A], \\
(\bot: \bot), (\bot:(\bot: \bot)), (A:(\bot: \bot)), (\bot:(A: \bot)), (A:(A: \bot)) \} \cup \ldots
\]

containing many more elements than in Example 1 because \( \bot \) elements are embedded.

- \( \text{Non-tail-strict}(A) \equiv \text{Univ}^{[\text{unit}A]} \)
Inference Rules of Programmatica logic
Constructors as Predicates

• Idea: Data constructors are “lifted” to act as predicate constructors

• Example:
  
x::: [] is the proposition “x has the value [] or else is undefined”

  x::: (P:Q) is the proposition “∃u,v. x has value (u:v) and u::: P and v::: Q or else x is undefined”
Rules in the style of a Sequent Calculus

• Right-introduction rules
  Example: \[ \Gamma \vdash h:::P \quad \Gamma \vdash t:::Q \]
  \[ \Gamma \vdash (h:t):::(P:Q) \]
  – Hypotheses make assertions about subterms of the subject term that appears on the right side of the conclusion

• Left-introduction rules
  Example: \[ h:::P, t:::Q \vdash \Delta \]
  \[ (h:t):::(P:Q) \vdash \Delta \]
  – Hypotheses make assumptions about subterms of a subject term that appears on the left side of the conclusion

Left introduction rules in a sequent style correspond to elimination rules in a natural deduction style.
Rules for Congruence Formulas

- Constructor application, right introduction
  \[
  \Gamma, (C, (\tau_1, \ldots, \tau_k)) \in \Sigma_k \quad \vdash \quad x_1 ::::: P_1^{\tau_1} \ldots \Gamma \vdash x_k ::::: P_k^{\tau_k}
  \]

- Constructor application, left introduction
  \[
  (C, (\tau_1, \ldots, \tau_k)) \in \Sigma_k, \quad x_1 ::::: P_1^{\tau_1}, \ldots, x_k ::::: P_k^{\tau_k} \vdash \Delta
  \]
  \[
  (C, (\tau_1, \ldots, \tau_k)) \in \Sigma_k, \quad C x_1 \ldots x_k ::::: C[\tau] P_1^{\tau_1} \ldots P_k^{\tau_k} \vdash \Delta
  \]
Abs traction and Application

• Abs traction (right introduction)

\[
\frac{\Gamma, x::: P \vdash e::: Q}{\Gamma \vdash \lambda x \rightarrow e::: (P \rightarrow Q)}
\]

– The arrow (→) is a predicate constructor symbol

• Abs traction (left introduction)

\[
\frac{\Gamma \vdash e::: P \quad \Gamma, f e::: Q \vdash \Delta}{\Gamma, f::: (P \rightarrow Q) \vdash \Delta}
\]

• Application (right introduction)

\[
\frac{\Gamma \vdash f::: (P \rightarrow Q) \quad \Gamma \vdash e::: P}{\Gamma \vdash f e::: Q}
\]
Properties of Recursively-Defined Functions — LFP Formulas

• A verification rule for LFP properties of a recursive function definition, let $m = M$

\[
\Gamma, m ::::: Univ \vdash M ::::: \$(P_1 \rightarrow H) \\
\Gamma, m ::::: (P_1 \lor P_2) \rightarrow \xi \vdash M ::::: \$(P_2 \rightarrow H) \\
\Gamma, m \models M \vdash m ::::: \$(P_1 \lor P_2 \rightarrow \text{Lfp } \xi \cdot H)
\]

- $P_1$ and $P_2$ are separation predicates

which partition the argument set into subsets on which $m$ is not recursively invoked (resp. is invoked) in $M$

- $\xi$ is a predicate variable that may occur only in $H$
Example: \textit{fact} yields a positive result on a domain of non-negative integers

\begin{itemize}
  \item \texttt{fact} = \( \lambda n \rightarrow \begin{cases} 
  1 & \text{if } n == 0 \\
  n \times \text{fact}(n-1) & \text{else}
  \end{cases} \) \hspace{1cm} (Haskell definition)

  \textbf{assert} \quad \text{fact} :::: \$(\geq 0) \rightarrow \$(\text{Lfp} \, \xi \cdot \$(==1) \lor \text{Geq} \, \xi)\\

  \textbf{property} \quad \text{Geq} \, P \equiv \{ x \mid \exists y \cdot x \geq y \land y :::: P \} \]

  \item \text{Separation predicates: } P1 \equiv \$(==0), \ P2 \equiv \$(>0), \ \text{support deductions of:} \\
  \text{fact} :: \text{Univ} \vdash (\lambda n \rightarrow \begin{cases} 
  1 & \text{if } n == 0 \\
  n \times \text{fact}(n-1) & \text{else}
  \end{cases}) \\
  \vdash \$(==0) \rightarrow \$(==1) \lor \text{Geq} \, \xi \\
  \text{and (using several facts about arithmetic)} \\
  \text{fact} :::: \$(\geq 0) \rightarrow \xi \vdash (\lambda n \rightarrow \begin{cases} 
  1 & \text{if } n == 0 \\
  n \times \text{fact}(n-1) & \text{else}
  \end{cases}) \\
  \vdash \$(>0) \rightarrow \$(==1) \lor \text{Geq} \, \xi \\
  \text{from which the assertion can be proved by the LFP rule}
\end{itemize}
Properties of Recursively-Defined Functions — GFP Formulas

• A verification rule for GFP properties of a recursive function definition, let $m = M$

\[
\Gamma \vdash M :::: H[Univ / \xi]
\]

\[
\Gamma, M :::: H \vdash m :::: \xi
\]

\[
\Gamma, m \equiv M \vdash m :::: Gfp\xi \bullet H
\]

where $\xi$ is a predicate variable that may occur only in $H$
Patterned Abs tractions

• Explicit abs traction over argument patterns
  – Extended with guarded expressions as the bodies of abs tractions
    (This is an orthogonal extension to Haskell — not part of the language)
• Function definitions, case expressions and let clauses can be defined in terms of patterned abs tractions
• The fatbar connective combines a sequence of patterned abs tractions into a composite, function-typed expression
  – Defined by interpreting patterned abs traction in the Maybe monad
Rules for a Patterned Abstraction

• **Successful match**

\[
\Gamma, x_1:::P_1,\ldots,x_n:::P_n \mid- g:::Q \\
\Gamma, e:::\pi(P_1,\ldots,P_n) \mid- (\lambda\pi(x_1,\ldots,x_n) \rightarrow g) e:::\pi\lambda Just(Q)
\]

where \(\pi\) represents a pattern with \(n\) variables

• **Match failure**

\[
\Gamma, e:::\pi Dom(\pi) \mid- (\lambda\pi(x_1,\ldots,x_n) \rightarrow g) e:::\pi Nothing
\]

where \(Dom(\pi)\) is the predicate satisfied by terms that do not match \(\pi\)
The fatbar connective

$(\|) \::: (a \rightarrow \text{Maybe } b) \rightarrow (a \rightarrow \text{Maybe } b) \rightarrow a \rightarrow \text{Maybe } b$

• Rules

\[
\begin{align*}
\Gamma \vdash g_1 \ e:::\$Nothing & \quad \Gamma \vdash g_2 \ e:::\$Q \\
\Gamma \vdash (g_1 \parallel g_2) \ e:::\$Q \\
\Gamma \vdash g_1 \ e:::\$Just(P) & \quad \Gamma \vdash (g_1 \parallel g_2) \ e:::\$Just(P)
\end{align*}
\]
Guarded Expressions

- Rules for expressions with guards
  - Maybe is a monadic type constructor

\[
\Gamma |- e :: P \quad \Gamma |- g :: $!(== True) \\
\Gamma |- g \rightarrow e :: $Just(P)
\]

where \( P :: Prop \)

\[
\Gamma |- g :: $!(== False) \\
\Gamma |- g \rightarrow e :: $Nothing
\]
Confirming property assertions in the Maybe monad

- Properties asserted in the Maybe monad are collected over branches of a case expression.
- But at the end of a list of case branches,
  - A strongly *Just*-prefixed property is equivalent to an ordinary predicate.

\[
\Gamma |- e :::: $Just(P) \\
\Gamma |- e :::: P
\]
Class Instances and Overloading

- Two kinds of overloading
  - Derived instances of an operator are language- (or implementation)-defined
    - Derived instances are generic functions
    - Derived instances satisfy a common law
  - Programmer-defined instances are particular
    - Instances have independent properties

- Overloading is resolved (logically) by typing
  - Use type-indexed predicates to specify properties
  - Give the meaning of an assertion at each instance of its index type
Type-Indexed Predicates

• Each predicate is annotated with a type formula
  – Indicates the type at which the predicate is interpreted
  – The predicate index on a formula must be compatible with the type of the expressions to which it applies
    • If e has type \( \tau \) then \( e ::: P^\tau \) has meaning

• Predicates in rules for generic operators may be indexed with a (qualified) type variable
  – For example, \( !(==0)^{\text{Num}} a => a \)

• Rules for specific operators may contain predicate expressions indexed by concrete types
Soundness of the Programatica logic

A sequent, $\Gamma \vdash e :::: P^\tau$, is valid if

For every semantic valuation (of term variables) such that all propositions of the context, $\Gamma$, are true, the conclusion $e :::: P^\tau$ is true (if $e$ has type $\tau$).

A criterion for soundness of an inference rule,

$\Gamma \vdash \text{Prop}_1 \ldots \Gamma \vdash \text{Prop}_n$

$\Gamma \vdash \text{Prop}$

For every context, $\Gamma$, such that the antecedents of the rule are valid sequents, the consequent is also valid.

$P$-logic is sound iff all of its inference rules are sound with respect to a semantics for Haskell.

Soundness proof is presented in a separate talk.
Tool Support for P-logic

• PFE, the Programatica Front-End tool
  – Parses and type-checks property assertions and declarations embedded in a Haskell program text
  – Interfaces with the PFE browser, which displays a text with embedded assertions
    • supports Haskell module structure
    • provides links to declarations of identifiers
  – Supports certificate management for asserted properties
    • automatically calculates and updates dependencies

• PFE is described in another talk (by Thomas Hallgren)
Conclusions

• \( P \)-logic meets its design objectives
  – Expresses properties of Haskell terms
    • Without translation or artificial coding
    • With modalities for both strict and non-strict functions and
      data constructors

• Its semantics is given in terms of a domain-theoretic model for Haskell
  – Semantics furnishes a reference for soundness of
    proof rules

• To be done:
  – Develop a verification server for \( P \)-logic assertions