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 μ-calculus
      - μ-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/existential properties of (unbounded) term structures

A Syntax of Formulas

Propositions:

M ::: P -- asserts that M satisfies P

- where M is a term and P is a predicate formula

M === N -- asserts (semantic) equality of M and N

Unary Predicates:

P ::= Univ -- the universal predicate
| Undef -- the predicate satisfied only by $\perp$
| $P_1 \wedge P_2$ -- a conjunctive predicate formula
| $P_1 \vee P_2$ -- a disjunctive predicate formula
| $P_1 \rightarrow P_2$ -- an "arrow" formula
| \$P -- a strong predicate (requires well-definedness)
| Lfp $\xi \bullet$P -- a least fixpoint (LFP) formula
| Gfp $\xi \bullet$P -- a greatest fixpoint (GFP) formula
| C P_1 … P_k -- a term congruence formula

- Where C is a "lifted" data constructor of arity *k*

| !(<c) -- "lifted" sections
| {| pattern | *Prop* |} – 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.

      `Finite-list(A) = [P] P -> (A -> P -> P) -> 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 $M ::: C\ P_1 \ldots P_{\boldsymbol{k}}$ is equivalent to:

  $\exists\ N_1 \ldots N_{\boldsymbol{k}} \bullet M = C\ N_1 \ldots N_{\boldsymbol{k}} \wedge (N_1 ::: P_1) \wedge \ldots \wedge (N_k ::: P_k)$

  - where C is a data constructor of arity *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 $f :: \tau_1 \rightarrow \tau_2$

f ::: $\$(\$Univ \rightarrow \$Univ)$	asserts that f is total
f ::: $UnDef \rightarrow UnDef$	asserts that f is strict
f ::: $Univ \rightarrow \$Univ$	asserts that f is a constant fn (since f is presumed continuous)

Predicate Formulas

- Propositional connectives are lifted to connectives of unary predicate formulas

$$x ::: (P \wedge Q) \equiv_{def} x ::: P \wedge x ::: Q$$

$$x ::: (P \vee Q) \equiv_{def} x ::: P \vee x ::: Q$$

$$x ::: \$(P \wedge Q) \equiv_{def} x ::: \$P \wedge x ::: \$Q$$

$$x ::: \$(P \vee Q) \equiv_{def} x ::: \$P \vee x ::: \$Q$$

$$x ::: \neg P \equiv_{def} \neg (x ::: P) \vee x ::: UnDef$$

Recursively Defined Predicates

- μ-calculus extends first-order logic with least and greatest fixed-point formulas
  - Expresses properties asserted over the extent of a data structure
- Examples:
  - $\mathit{Finite\text{-}list} = \mathrm{Lfp}\ \xi \bullet [\,] \vee (\mathit{Univ} : \$\xi)$
  - $\mathit{Head\text{-}strict\text{-}list} = \mathrm{Lfp}\ \xi \bullet [\,] \vee (\$\mathit{Univ} : \xi)$
  - $\mathit{Infinite\text{-}Stream} = \mathrm{Gfp}\ \xi \bullet (\mathit{Univ} : \$\xi)$

Example 1: length of lists is additive

- Functions as defined in Haskell

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

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

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

  – Similarly,

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

  – We assert the following

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:

genPR :: (a -> Bool) -> (a -> a) -> c -> (a -> c -> c) -> a -> c

genPR p b g h x = **if** p x **then** g **else** h x (genPR p b g h (b x))

  - A factorial function:

fact :: Integer -> Integer

fact = genPR eq0 (subtract 1) 1 (*)

  - We assert the following

property $x \geq 0 \Rightarrow$ fact x = x!

where 0! = 1

(x+1)! = (x+1)*x!

Proved using an inductive proof rule for genPR

This rule requires that a set well-ordered by p and b be specified:

WO p b = Lfp $\eta \bullet$ {| x | p x \$=== True $\vee$ (b x ::: \$$\eta$) |}

Example 3: Ordered insertion in a list

- Functions as defined in 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:

```
insert a = \_ys -> case _ys of
                     [ ] -> [a]
                     (y :_ ) | a < y -> a : ys
                             | a == y -> ys
                     (y : ys) -> y : insert a ys
```

  – We assert the following

assert All xs • xs ::: $\square \$Univ \Rightarrow$ insert a xs ::: !(<a) **unless** !(==a)

where P **unless** Q = Gfp $\xi \bullet (\$Q : Univ) \vee (\$P : \$\xi)$

and $\square$P = Gfp $\xi \bullet [\,] \vee (P : \$\xi)$

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 τ (an *ideal**)

$C_{\lceil \tau \rceil}$ is the interpretation of the constant symbol C in the type τ

$[\![P]\!]^{\tau}$ is the ideal of domain elements $\{t \in \lceil \tau \rceil \mid t \text{ satisfies } P\}$, where P is an unstrengthened predicate

$[\![\$P]\!]^{\tau}$ is the set of elements $[\![P]\!]^{\tau} - \{\bot\}$

* (Recall that an *ideal* poset is downward-closed and contains limits of its finite directed subsets.)

For Example: Distinguished Predicates

Strong modality

$[\![\$Univ]\!]^{\tau} = \lceil \tau \rceil - \{\perp\}$

$[\![\$UnDef]\!]^{\tau} = \{\ \}$

Weak modality

$[\![Univ]\!]^{\tau} = \lceil \tau \rceil$

$[\![UnDef]\!]^{\tau} = \{\perp\}$

Predicates derived from Sections

- Equality comparisons with constants

  $[\![!(==a)]\!]^{\tau} = \{x \in \lceil \tau \rceil \mid x = a_{\lceil \tau \rceil} \vee x = \bot\}$

- Ordering relations

  $[\![!(<a)]\!]^{\tau} = \{x \in \lceil \tau \rceil \mid x <_{\lceil \tau \rceil} a_{\lceil \tau \rceil} \vee x = \bot\}$

  where τ is an instance of the *Ord* type class

Term Congruence Predicates

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

$(C, (\tau_1, \ldots, \tau_k)) \in \Sigma^\tau{}_k$ **=>**

$$[\![C\ P_1 \ldots P_k]\!]^\tau =$$

$$\{C_{\lceil\tau\rceil}\, x_1 \ldots x_k \mid x_1 \in [\![P_1]\!]^{\tau_1} \wedge \ldots \wedge x_k \in [\![P_k]\!]^{\tau_k}\} \cup \{\perp\}$$

Arrow Predicates

$$[\![P \to Q]\!]^{\tau_1 \to \tau_2} =$$

$$\{f \in \lceil\tau_1\rceil \to \lceil\tau_2\rceil \mid \forall x \in [\![P]\!]^{\tau_1} \bullet f\ x \in [\![Q]\!]^{\tau_2}\} \cup \{\perp\}$$

Conjunction and Disjunction

$$[\![P_1 \wedge P_2]\!]^{\tau} = [\![P_1]\!]^{\tau_1} \cap [\![P_2]\!]^{\tau}$$

$$[\![P_1 \vee P_2]\!]^{\tau} = [\![P_1]\!]^{\tau_1} \cup [\![P_2]\!]^{\tau}$$

The Equality Predicate

$$[\![(==)]\!]^{\tau} = \{(u,v) \mid u \in \lceil \tau \rceil \wedge v \in \lceil \tau \rceil \wedge u = v\}$$

$$[\![(\$==)]\!]^{\tau} = \{(u,v) \mid u \in \lceil \tau \rceil \wedge v \in \lceil \tau \rceil \wedge u = v \wedge u \neq \bot\}$$

Fixed-Point Formulas

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

LFP: $$[\![\mathsf{Lfp}\,\xi \bullet H]\!]^{\tau} = \bigcup_{j=0}^{\infty} [\![H^{j}]\!]^{\tau}$$

where $H^{0} = \mathit{UnDef}$

$H^{j+1} = H\,[H^{j}/\xi]$

GFP: $$[\![\mathsf{Gfp}\,\xi \bullet H]\!]^{\tau} = \bigcap_{j=0}^{\infty} [\![H^{j}]\!]^{\tau}$$

where $H^{0} = \mathit{Univ}$

$H^{j+1} = H\,[H^{j}/\xi]$

Example 1: Tail-strict lists

- Consider LFP formula

  $Strict\text{-}list(A) \equiv \text{Lfp}\ \xi \bullet \mathbf{[\ ]} \vee (\$A \mathbin{\mathbf{:}} \$\xi)$

  where **`data`** $unitA = A$

- The interpretation of *Strict-list(A)* is

  $\{\perp\} \cup \{\perp, \mathbf{[\,]}\} \cup \{\perp, \mathbf{[\,]}, \mathbf{[}A\mathbf{]}\} \cup \{\perp, \mathbf{[\,]}, \mathbf{[}A\mathbf{]}, \mathbf{[}A,A\mathbf{]}\} \cup \ldots$

This is the representation of a *flat* subdomain (the $\perp$ element is never embedded in a list structure)

Example 2: Non-tail-strict lists

- Consider LFP formula

  $Non\text{-}strict(A) \equiv \text{Lfp } \xi \bullet \mathbf{[\,]} \vee (A \mathbin{\mathbf{:}} \xi)$

  where **`data`** $unitA = A$

- The interpretation of *Non-strict(A)* is

  $\{\perp\} \cup \{\perp, \mathbf{[\,]}, (\perp{:}\perp), (A{:}\perp)\} \cup \{\perp, \mathbf{[\,]}, \mathbf{[}\perp\mathbf{]}, \mathbf{[}A\mathbf{]},$
  $(\perp{:}\perp), (\perp{:}(\perp{:}\perp)), (A{:}(\perp{:}\perp)), (\perp{:}(A{:}\perp)), (A{:}(A{:}\perp))\} \cup \ldots$

  containing many more elements than in Example 1 because $\perp$ elements are embedded

- $Non\text{-}tail\text{-}strict(A) \equiv Univ^{[unitA]}$

Inference Rules of Programatica 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 "$\exists u,v.$ 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: $$\frac{\Gamma \vdash h:::P \quad \Gamma \vdash t:::Q}{\Gamma \vdash (h:t):::(P:Q)}$$

  – Hypotheses make assertions about subterms of the subject term that appears on the right side of the conclusion

- Left-introduction rules

  Example: $$\frac{h:::P,\, t:::Q \ \vdash \ \Delta}{(h:t):::(P:Q) \vdash \Delta}$$

  – 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

$$\frac{\Gamma, (C,(\tau_1,\ldots,\tau_k)) \in \Sigma_k^{\tau} \quad \vdash x_1 ::: P_1^{\tau_1} \ldots \quad \Gamma \vdash x_k ::: P_k^{\tau_k}}{\Gamma, (C,(\tau_1,\ldots,\tau_k)) \in \Sigma_k^{\tau} \vdash C\, x_1 \ldots x_k ::: C_{\lceil \tau \rceil} P_1^{\tau_1} \ldots P_k^{\tau_k}}$$

- Constructor application, left introduction

$$\frac{(C,(\tau_1,\ldots,\tau_k)) \in \Sigma_k^{\tau}, \qquad x_1 ::: P_1^{\tau_1},\ldots, x_k ::: P_k^{\tau_k} \vdash \Delta}{(C,(\tau_1,\ldots,\tau_k)) \in \Sigma_k^{\tau}, C\, x_1 \ldots x_k ::: C_{\lceil \tau \rceil} P_1^{\tau_1} \ldots P_k^{\tau_k} \vdash \Delta}$$

Abstraction and Application

- Abstraction (right introduction)

$$\frac{\Gamma, x ::: P \vdash e ::: Q}{\Gamma \vdash \ \lambda x \text{->} e ::: \$(P \to Q)}$$

  - **The arrow (→) is a predicate constructor symbol**

- Abstraction (left introduction)

$$\frac{\Gamma \vdash e ::: P \quad \Gamma, f\, e ::: Q \vdash \Delta}{\Gamma, f ::: \$(P \to Q) \vdash \Delta}$$

- Application (right introduction)

$$\frac{\Gamma \vdash f ::: \$(P \to Q) \quad \Gamma \vdash e ::: P}{\Gamma \ \vdash \ f\, e ::: Q}$$

Properties of Recursively-Defined Functions — LFP Formulas

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

$$\frac{\Gamma, m ::: Univ \vdash M ::: \$(P_1 \rightarrow H) \qquad \Gamma, m ::: (P_1 \vee P_2) \rightarrow \xi \vdash M ::: \$(P_2 \rightarrow H)}{\Gamma, m === M \vdash m ::: \$(P_1 \vee P_2 \rightarrow \mathsf{Lfp}\,\xi \bullet H)}$$

  – 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*

  – ξ is a predicate variable that may occur only in *H*

Example: *fact* yields a positive result on a domain of non-negative integers

- $fact = \lambda n$ -> if n == 0 then 1 (*Haskell definition*)
  else $n * fact\ (n - 1)$

  assert $fact$ ::: $\$!(\geq 0) \rightarrow \$(\mathrm{Lfp}\ \xi \bullet \$!(==1) \vee Geq\ \xi)$

  property $Geq\ P = \{x \mid \exists y \bullet x \geq y \wedge y ::: P\}$
- Separation predicates: $P1 \equiv \$!(==0)$, $P2 \equiv \$!(>0)$, support deductions of:

$$fact :: Univ \vdash (\lambda n \text{ -> if } n == 0 \text{ then } 1 \text{ else } n * fact(n-1))$$

Separation constraint

$$::: \$!(== 0) \rightarrow \$(!(== 1) \vee Geq\ \xi)$$

and (using several facts about arithmetic)

$$fact ::: \$!(\geq 0) \rightarrow \xi \vdash (\lambda n \text{ -> if } n == 0 \text{ then } 1 \text{ else } n * fact(n-1))$$

$$::: \$!(> 0) \rightarrow \$(!(== 1) \vee Geq\ \xi)$$

Separation constraint

from which the assertion can be proved by the LFP rule

Properties of Recursively-Defined Functions — GFP Formulas

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

$$\frac{\Gamma \vdash M ::: \$H[Univ / \xi] \qquad \Gamma, M ::: \$H \vdash m ::: \xi}{\Gamma, m === M \vdash m ::: \mathsf{Gfp}\,\xi \bullet H}$$

where ξ is a predicate variable that may occur only in H

Patterned Abstractions

- Explicit abstraction over argument patterns
  - Extended with guarded expressions as the bodies of abstractions

  (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 abstractions
- The fatbar connective combines a sequence of patterned abstractions into a composite, function-typed expression
  - Defined by interpreting patterned abstraction in the *Maybe* monad

Rules for a Patterned Abstraction

- Successful match

$$\frac{\Gamma, x_1 ::: P_1, \ldots, x_n :: P_n \vdash g ::: Q}{\Gamma,\ e ::: \$\pi(P_1, \ldots, P_n) \vdash (\lambda\pi(x_1, \ldots, x_n) -> g)\, e ::: \$Just(Q)}$$

where π represents a pattern with *n* variables

- Match failure

$$\Gamma, e ::: \$\overline{Dom(\pi)} \vdash (\lambda\pi(x_1, \ldots, x_n) -> g)\, e ::: \$Nothing$$

where *Dom* (π) is the predicate satisfied by terms that do not match π

The fatbar connective

(‖) :: (a → Maybe b) → (a → Maybe b) → a → Maybe b

- Rules

$$\frac{\Gamma \vdash g_1\, e ::: \$Nothing \qquad \Gamma \vdash g_2\, e ::: Q}{\Gamma \vdash (g_1 \| g_2)\, e ::: Q}$$

$$\frac{\Gamma \vdash g_1\, e ::: \$Just(P)}{\Gamma \vdash (g_1 \| g_2)\, e ::: \$Just(P)}$$

Guarded Expressions

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

$$\frac{\Gamma \vdash e ::: P \qquad \Gamma \vdash g ::: \$!(== True)}{\Gamma \vdash g \text{ -> } e ::: \$Just(P)}$$

where *P* :: *Prop*

$$\frac{\Gamma \vdash g ::: \$!(== False)}{\Gamma \vdash g \text{ -> } e ::: \$Nothing}$$

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

$$\frac{\Gamma \vdash e ::: \$Just(P)}{\Gamma \vdash e ::: P}$$

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 τ then e ::: P^{τ} has meaning
- Predicates in rules for generic operators may be indexed with a (qualified) type variable
  - For example, $!(==0)^{Num\ a\ =>\ a}$
- 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, Γ, are true, the conclusion $e :::P^{\tau}$ is true (if e has type τ).

A criterion for soundness of an inference rule,

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

For every context, Γ, 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