FACULTY OF SCIENCE

Bart Jacobs

Structuring Computations

Radboud University Nijmegen

Contents

Contents

Contents

No explicit message;
some type/object-related
topics that I like;
and you too, hopefully!

I. Sneak preview

Purely functional programs

Purely functional programs

Writing X for the type of inputs, Y for outputs . . .

Purely functional programs

Writing X for the type of inputs, Y for outputs . . .

. . . a functional program from X to Y is simply a function

$$X \longrightarrow Y$$

Imperative, state-based programs

Imperative, state-based programs

Writing S for the type of states ...

Imperative, state-based programs

Writing S for the type of states . . .

. . . an ***imperative*** program is:

$$X \times S \longrightarrow Y \times S$$

Imperative, state-based programs

Writing S for the type of states . . .

. . . an ***imperative*** program is:

$$X \times S \longrightarrow Y \times S$$

Or, equivalently,

$$X \longrightarrow (Y \times S)^S$$

Imperative, state-based programs

Writing S for the type of states . . .

. . . an ***imperative*** program is:

$$X \times S \longrightarrow Y \times S$$

Or, equivalently,

$$X \longrightarrow (Y \times S)^S$$

Involving the ***State Monad*** $Y \longmapsto (Y \times S)^S$

Reactive, stream-based programs

Reactive, stream-based programs

A ***reactive*** program is:

$$X^{\mathbb{N}} \longrightarrow Y^{\mathbb{N}}$$

Reactive, stream-based programs

A ***reactive*** program is:

$$X^{\mathbb{N}} \longrightarrow Y^{\mathbb{N}}$$

Or, equivalently,

$$X^{\mathbb{N}} \times \mathbb{N} \longrightarrow Y$$

Reactive, stream-based programs

A ***reactive*** program is:

$$X^{\mathbb{N}} \longrightarrow Y^{\mathbb{N}}$$

Or, equivalently,

$$X^{\mathbb{N}} \times \mathbb{N} \longrightarrow Y$$

Involving the ***Stream Comonad*** $X \longmapsto X^{\mathbb{N}} \times \mathbb{N}$

Quantum program

Quantum program

A possible ***quantum*** program is:

$$X \times X \longrightarrow [0,1]^{(Y \times Y)}$$

Quantum program

A possible ***quantum*** program is:

$$X \times X \longrightarrow [0,1]^{(Y \times Y)}$$

It is a "superoperator" on "density matrices" (or quantum states)—after Vizotto, Altenkirch, Sabry

Quantum program

A possible ***quantum*** program is:

$$X \times X \longrightarrow [0,1]^{(Y \times Y)}$$

It is a “superoperator” on “density matrices” (or quantum states)—after Vizotto, Altenkirch, Sabry

It forms an example of an ***Arrow***: computations with unit and composition.

Overview

Overview

- **Functional:** $X \longrightarrow Y$

Overview

- **Functional:** $X \longrightarrow Y$
- **Imperative:** $X \longrightarrow T(Y)$, with T monad (including Java programs)

Overview

- **Functional:** $X \longrightarrow Y$
- **Imperative:** $X \longrightarrow T(Y)$, with T monad (including Java programs)
- **Reactive:** $G(X) \longrightarrow Y$, with G comonad

Overview

- **Functional:** $X \longrightarrow Y$
- **Imperative:** $X \longrightarrow T(Y)$, with T monad (including Java programs)
- **Reactive:** $G(X) \longrightarrow Y$, with G comonad
- **Quantum:** $A(X, Y)$, with A “arrow”

II. Comonads

Comonads for computations

Comonads for computations

- Monads are well-established in functional programming & language semantics

Comonads for computations

- Monads are well-established in functional programming & language semantics
- But little attention for the dual notion of comonad . . .

Comonads for computations

- Monads are well-established in functional programming & language semantics
- But little attention for the dual notion of comonad . . .
- . . . until Uustalu & Vene recently used them for structuring reactive/dataflow programming—building on Brookes & Geva

Comonads for computations

- Monads are well-established in functional programming & language semantics
- But little attention for the dual notion of comonad . . .
- . . . until Uustalu & Vene recently used them for structuring reactive/dataflow programming—building on Brookes & Geva
- **Slogan:** monads structure output, comonads structure input

Comonad structure

Comonad structure

- ***Categorically:*** endofunctor $G\colon \mathbb{C} \longrightarrow \mathbb{C}$ with two natural transformations $\varepsilon\colon G \Rightarrow \mathrm{Id}$ and $\delta\colon G \Rightarrow G^2$ satisfying standard equations

Comonad structure

- ***Categorically:*** endofunctor $G \colon \mathbb{C} \to \mathbb{C}$ with two natural transformations $\varepsilon \colon G \Rightarrow \mathrm{Id}$ and $\delta \colon G \Rightarrow G^2$ satisfying standard equations
- ***Computationally:*** Type operator G with
  - *coreturn* $: GX \longrightarrow X$
  - *cobind* $: (GX \to Y) \longrightarrow (GX \to GY)$

  satisfying suitable equations

Comonad structure

- ***Categorically:*** endofunctor $G\colon \mathbb{C} \to \mathbb{C}$ with two natural transformations $\varepsilon\colon G \Rightarrow \mathrm{Id}$ and $\delta\colon G \Rightarrow G^2$ satisfying standard equations
- ***Computationally:*** Type operator G with
  - *coreturn* $: GX \longrightarrow X$
  - *cobind* $: (GX \to Y) \longrightarrow (GX \to GY)$

  satisfying suitable equations
- ***Logically:*** structure for weakening and contraction (like bang ! in linear logic)

Comonad example

Comonad example

- Mapping $X \longmapsto X^{\mathbb{N}} \times \mathbb{N}$

Comonad example

- Mapping $X \longmapsto X^{\mathbb{N}} \times \mathbb{N}$
- Input streams with past / current / future:

$$x_0, x_1, \ldots, x_{n-1}, \boxed{x_n}, x_{n+1}, x_{n+2}, \ldots$$

Comonad example

- Mapping $X \longmapsto X^{\mathbb{N}} \times \mathbb{N}$
- Input streams with past / current / future:

$$x_0, x_1, \ldots, x_{n-1}, \boxed{x_n}, x_{n+1}, x_{n+2}, \ldots$$

- Counit / coreturn: $X^{\mathbb{N}} \times \mathbb{N} \longrightarrow X$

$$(\alpha, n) \longmapsto \alpha(n)$$

Comonad example

- Mapping $X \longmapsto X^{\mathbb{N}} \times \mathbb{N}$
- Input streams with past / current / future:

$$x_0, x_1, \ldots, x_{n-1}, \boxed{x_n}, x_{n+1}, x_{n+2}, \ldots$$

- Counit / coreturn: $X^{\mathbb{N}} \times \mathbb{N} \longrightarrow X$

$$(\alpha, n) \longmapsto \alpha(n)$$

- Delta: $X^{\mathbb{N}} \times \mathbb{N} \longrightarrow (X^{\mathbb{N}} \times \mathbb{N})^{\mathbb{N}} \times \mathbb{N}$

$$(\alpha, n) \longmapsto (\lambda m{:}\,\mathbb{N}.\,(\alpha, m), n)$$

coKleisli category of computations

coKleisli category of computations

- coKleisli maps $X^{\mathbb{N}} \times \mathbb{N} \longrightarrow Y$ form a category

coKleisli category of computations

- coKleisli maps $X^{\mathbb{N}} \times \mathbb{N} \longrightarrow Y$ form a category
- Identity via coreturn; composition via delta/cobind

coKleisli category of computations

- coKleisli maps $X^{\mathbb{N}} \times \mathbb{N} \longrightarrow Y$ form a category
- Identity via coreturn; composition via delta/cobind
- Gives output in Y for completely given input stream of X's

coKleisli category of computations

- coKleisli maps $X^{\mathbb{N}} \times \mathbb{N} \longrightarrow Y$ form a category
- Identity via coreturn; composition via delta/cobind
- Gives output in Y for completely given input stream of X's
- Basis for dataflow calculus by Uustalu & Vene
  (like in Lustre, Lucid)

Discrete time signals

Discrete time signals

Three basic comonads:

Discrete time signals

Three basic comonads:

$$X^{\star} \times X \xleftarrow[\text{no future}]{\text{causality}} X^{\mathbb{N}} \times \mathbb{N} \xrightarrow[\text{no past}]{\text{anti-causality}} X^{\mathbb{N}}$$

$$(\langle \alpha(0), \ldots, \alpha(n-1) \rangle, \alpha(n)) \longleftarrow\!\shortmid (\alpha, n) \longmapsto \lambda m.\, \alpha(n+m)$$

Discrete time signals

Three basic comonads:

$$\begin{tikzcd}[column sep=huge] X^{\star} \times X & X^{\mathbb{N}} \times \mathbb{N} \arrow[l, "\text{causality}"', "\text{no future}"] \arrow[r, "\text{anti-causality}", "\text{no past}"'] & X^{\mathbb{N}} \end{tikzcd}$$

$$(\langle \alpha(0),\ldots,\alpha(n-1)\rangle,\alpha(n)) \longmapsfrom (\alpha,n) \longmapsto \lambda m.\, \alpha(n+m)$$

with “comonad homomorphisms” between them

Continuous time signals

Continuous time signals

Analogues fundamental diagram of comonads:

Continuous time signals

Analogues fundamental diagram of comonads:

$$\coprod_{t \in [0,\infty)} X^{[0,t)} \times X \longleftarrow X^{[0,\infty)} \times [0,\infty) \longrightarrow X^{[0,\infty)}$$

Continuous time signals

Analogues fundamental diagram of comonads:

$$\coprod_{t\in[0,\infty)} X^{[0,t)} \times X \longleftarrow X^{[0,\infty)} \times [0,\infty) \longrightarrow X^{[0,\infty)}$$

where:

$$\coprod_{t\in[0,\infty)} X^{[0,t)} \times X \;\cong\; \coprod_{t\in[0,\infty)} X^{[0,t]} \;\cong\; X^{[0,1]} \times [0,\infty)$$

III. Arrows

Arrow overview

Arrow overview

- Introduced in Haskell by Hughes in 2000, as common interface extending monads (parser as main example)

Arrow overview

- Introduced in Haskell by Hughes in 2000, as common interface extending monads (parser as main example)
- Binary type operation $A(-,+)$ with three operations: arr, $\ggg$, first.

Arrow overview

- Introduced in Haskell by Hughes in 2000, as common interface extending monads (parser as main example)
- Binary type operation $A(-,+)$ with three operations: arr, $\ggg$, first.
- Folklore claim: Arrows are Freyd categories (Power & Robinson'99)

Arrow overview

- Introduced in Haskell by Hughes in 2000, as common interface extending monads (parser as main example)
- Binary type operation $A(-,+)$ with three operations: arr, $\ggg$, first.
- Folklore claim: Arrows are Freyd categories (Power & Robinson'99)
- Recently substantiated by first describing arrows as ***monoids*** in a category of bifunctors $\mathbb{C}^{\mathrm{op}} \times \mathbb{C} \rightarrow$ **Sets**

Arrow in Haskell

Arrow in Haskell

Introduced as type class:

Arrow in Haskell

Introduced as type class:

class Arrow A where

  arr :: $(X \to Y) \to$ A $X\,Y$

  $(\ggg)$ :: A $X\,Y \to$ A $Y\,Z \to$ A $X\,Z$

  first :: A $X\,Y \to$ A $(X, Z)\,(Y, Z)$

Arrow in Haskell

Introduced as type class:

class Arrow A where

arr :: $(X \to Y) \to$ A $X\,Y$

$(\ggg)$ :: A $X\,Y \to$ A $Y\,Z \to$ A $X\,Z$

first :: A $X\,Y \to$ A $(X, Z)\,(Y, Z)$

Which should satisfy 8 equations, such as:

$$\begin{array}{rcl} (a \ggg b) \ggg c & = & a \ggg (b \ggg c) \\ a \ggg \mathsf{arr}(1) & = & a \\ \mathsf{first}(\mathsf{arr}(f)) & = & \mathsf{arr}(f \times 1), \quad \textit{etc} \end{array}$$

Arrow examples

Arrow examples

- $(X, Y) \longmapsto (X \to T(Y))$, for T monad
  $(X, Y) \longmapsto (G(X) \to Y)$, for G comonad

Arrow examples

- $(X, Y) \longmapsto (X \to T(Y))$, for T monad
  $(X, Y) \longmapsto (G(X) \to Y)$, for G comonad
- $(X, Y) \longmapsto (X \times X \to [0, 1]^{(Y \times Y)})$ for quantum computation

Arrow examples

- $(X, Y) \longmapsto (X \to T(Y))$, for T monad
  $(X, Y) \longmapsto (G(X) \to Y)$, for G comonad
- $(X, Y) \longmapsto (X \times X \to [0, 1]^{(Y \times Y)})$ for quantum computation
- $(X, Y) \longmapsto (X^{\mathbb{N}} \to \mathcal{P}(Y^{\mathbb{N}}))$ for "non-deterministic dataflow"

Arrow examples

- $(X, Y) \longmapsto (X \to T(Y))$, for T monad
  $(X, Y) \longmapsto (G(X) \to Y)$, for G comonad
- $(X, Y) \longmapsto (X \times X \to [0, 1]^{(Y \times Y)})$ for quantum computation
- $(X, Y) \longmapsto (X^{\mathbb{N}} \to \mathcal{P}(Y^{\mathbb{N}}))$ for "non-deterministic dataflow"
- $$(X, Y) \longmapsto \; (2 \times S^{\star}) \times ((S^{\star} \times X) \to (1 + (S^{\star} \times Y)))$$
  for Swierstra-Duponcheel parser that motivated Hughes

Arrows, categorically

Arrows, categorically

- A is functorial: for $f\colon X' \to X$ and $g\colon Y \to Y'$,

$$\begin{array}{ccc} A(X,Y) & \xrightarrow{\quad A(f,g) \quad} & A(X',Y') \\ a & \longmapsto & \mathsf{arr}(f) \ggg a \ggg \mathsf{arr}(g) \end{array}$$

Arrows, categorically

- A is functorial: for $f\colon X' \to X$ and $g\colon Y \to Y'$,

$$\begin{array}{ccc} A(X,Y) & \xrightarrow{\quad A(f,g) \quad} & A(X',Y') \\ a & \longmapsto & \mathsf{arr}(f) \ggg a \ggg \mathsf{arr}(g) \end{array}$$

- arr: $(+)^{(-)} \to A(-,+)$ is natural transformation (natro, for short)

Arrows, categorically

- A is functorial: for $f\colon X' \to X$ and $g\colon Y \to Y'$,

$$\begin{array}{ccc} A(X,Y) & \xrightarrow{\quad A(f,g) \quad} & A(X',Y') \\ a & \longmapsto & \mathrm{arr}(f) \ggg a \ggg \mathrm{arr}(g) \end{array}$$

- arr: $(+)^{(-)} \to A(-,+)$ is natural transformation (natro, for short)
- $\ggg$ is natro $A \otimes A \to A$, for tensor product of distributors / profunctors

Arrows, categorically

- A is functorial: for $f\colon X' \to X$ and $g\colon Y \to Y'$,

$$\begin{array}{ccc} A(X,Y) & \xrightarrow{\qquad A(f,g) \qquad} & A(X',Y') \\ a & \longmapsto & \mathsf{arr}(f) \ggg a \ggg \mathsf{arr}(g) \end{array}$$

- arr: $(+)^{(-)} \to A(-,+)$ is natural transformation (natro, for short)
- $\ggg$ is natro $A \otimes A \to A$, for tensor product of distributors / profunctors
- first corresponds to “internal strength”

Excurs: monoid in a category

Excurs: monoid in a category

- Standardly, a monoid is a set M with associative $m\colon M \times M \to M$ and two-sided unit $e\colon 1 \to M$

Excurs: monoid in a category

- Standardly, a monoid is a set M with associative $m\colon M \times M \to M$ and two-sided unit $e\colon 1 \to M$
- Can be formulated in category with finite products $(1, \times)$: equations become diagrams

Excurs: monoid in a category

- Standardly, a monoid is a set M with associative $m\colon M \times M \longrightarrow M$ and two-sided unit $e\colon 1 \longrightarrow M$
- Can be formulated in category with finite products $(1, \times)$: equations become diagrams
- No projections/diagonals needed: also in monoidal category with $(I, \otimes)$. Eg.

$$\begin{tikzcd}
M \otimes M \arrow[d, "m"'] & M \otimes I \arrow[l, "1 \otimes e"'] & M \arrow[l, "\cong"'] \arrow[r, "\cong"] \arrow[dll, equal] \arrow[drr, equal] & I \otimes M \arrow[r, "e \otimes 1"] & M \otimes M \arrow[d, "m"] \\
M & & & & M
\end{tikzcd}$$

Excurs: monads are monoids

Excurs: monads are monoids

- The functor category $\mathbb{C}^{\mathbb{C}}$ is monoidal:

$$F \otimes G = F \circ G \qquad\qquad I = \mathrm{Id}$$

Excurs: monads are monoids

- The functor category $\mathbb{C}^{\mathbb{C}}$ is monoidal:

$$F \otimes G \;=\; F \circ G \qquad\qquad I \;=\; \mathrm{Id}$$

- A monoid in $\mathbb{C}^{\mathbb{C}}$ is a functor $M\colon \mathbb{C} \to \mathbb{C}$ with natros:

$$\begin{tikzcd} M \otimes M \arrow[r, "\mu"] \arrow[d, equal] & M & \mathrm{Id} \arrow[l, "\eta"'] \\ M \circ M & & \end{tikzcd}$$

satisfying the monoid equations

Excurs: monads are monoids

- The functor category $\mathbb{C}^{\mathbb{C}}$ is monoidal:

$$F \otimes G \;=\; F \circ G \qquad\qquad I \;=\; \mathrm{Id}$$

- A monoid in $\mathbb{C}^{\mathbb{C}}$ is a functor $M\colon \mathbb{C} \to \mathbb{C}$ with natros:

$$\begin{tikzcd} M \otimes M \arrow[r, "\mu"] \arrow[d, equal] & M & \mathrm{Id} \arrow[l, "\eta"'] \\ M \circ M & & \end{tikzcd}$$

satisfying the monoid equations

- A monoid in $\mathbb{C}^{\mathbb{C}}$ is precisely a monad!

Arrows are also monoids

Arrows are also monoids

- Arrows are monoids in category of bifunctors $\mathbb{C}^{\mathrm{op}} \times \mathbb{C} \longrightarrow \mathbf{Sets}$

Arrows are also monoids

- Arrows are monoids in category of bifunctors $\mathbb{C}^{\mathrm{op}} \times \mathbb{C} \to \mathbf{Sets}$
- Tensor $\otimes$ more complicated, with exponentiation/hom as unit

Arrows are also monoids

- Arrows are monoids in category of bifunctors $\mathbb{C}^{\mathrm{op}} \times \mathbb{C} \to$ **Sets**
- Tensor $\otimes$ more complicated, with exponentiation/hom as unit
- Allows for precise comparison with Freyd categories
  (bijective correspondence)

Arrows are also monoids

- Arrows are monoids in category of bifunctors $\mathbb{C}^{\mathrm{op}} \times \mathbb{C} \to \mathbf{Sets}$
- Tensor $\otimes$ more complicated, with exponentiation/hom as unit
- Allows for precise comparison with Freyd categories
(bijective correspondence)
- Details in Heunen & Jacobs, MFPS'06.

Arrows, intuitively

Arrows, intuitively

- Most fundamental mathematical structure in computing?

Arrows, intuitively

- Most fundamental mathematical structure in computing?
- Monoid $(A, ; , \mathsf{skip})$ of programs/actions $A \in$ **Sets** with sequential composition

Arrows, intuitively

- Most fundamental mathematical structure in computing?
- Monoid $(A, ; , \mathsf{skip})$ of programs/actions $A \in$ **Sets** with sequential composition
- Adding input and output makes $A(-, +)$ binary operator

Arrows, intuitively

- Most fundamental mathematical structure in computing?
- Monoid $(A, ;, \mathrm{skip})$ of programs/actions $A \in$ **Sets** with sequential composition
- Adding input and output makes $A(-,+)$ binary operator
- Hence carrier A becomes bifunctor $\mathbb{C}^{\mathrm{op}} \times \mathbb{C} \to$ **Sets**

Arrows, intuitively

- Most fundamental mathematical structure in computing?
- Monoid $(A, ;, \mathrm{skip})$ of programs/actions $A \in$ **Sets** with sequential composition
- Adding input and output makes $A(-, +)$ binary operator
- Hence carrier A becomes bifunctor $\mathbb{C}^{\mathrm{op}} \times \mathbb{C} \to$ **Sets**
- Keeping the monoid structure leads to Hughes' **Arrow**

IV. Monads

Monad overview

Monad overview

- Introduced by Moggi (1991), popularised in functional programming by Wadler

Monad overview

- Introduced by Moggi (1991), popularised in functional programming by Wadler
- for structuring outputs / computational effects

Monad overview

- Introduced by Moggi (1991), popularised in functional programming by Wadler
- for structuring outputs / computational effects
- Standard examples:
  - lift / maybe $1 + (-)$
  - exception $E + (-)$
  - list $(-)^\star$
  - state $(- \times S)^S$
  - non-determinism $\mathcal{P}$ (powerset)
  - probability $\mathcal{D}$ (distribution)

Java monad

Java monad

- Definition [Jacobs & Poll'03]:

$$J(X) \ = \ (1 + S \times X + S \times E)^S$$

Java monad

- Definition [Jacobs & Poll'03]:

$$J(X) \;=\; (1 + S \times X + S \times E)^S$$

- Combination of state, lift, exception monad

Java monad

- Definition [Jacobs & Poll'03]:

$$J(X) \;=\; (1 + S \times X + S \times E)^S$$

- Combination of state, lift, exception monad
- Actual "abnormal" termination in Java more complicated: exceptions, return, break, continue

Java monad

- Definition [Jacobs & Poll'03]:

$$J(X) \;=\; (1 + S \times X + S \times E)^S$$

- Combination of state, lift, exception monad
- Actual "abnormal" termination in Java more complicated: exceptions, return, break, continue
- Exception mechanism (plus logic) axiomatised as equaliser by [Schröder & Mossakowski]

Kleisli composition for Java monad

Kleisli composition for Java monad

- Kleisli composition for J is "argument evaluation, before use"
  (and not sequential composition ;)

Kleisli composition for Java monad

- Kleisli composition for J is "argument evaluation, before use"
  (and not sequential composition ;)
- For $a\colon X \to J(Y)$, and $p\colon Y \to J(Z)$,

$$
\begin{array}{rcl}
p \bullet a & = & \lambda x\colon X.\, \lambda s\colon S. \\
& & \quad \text{CASES } a\,x\,s \text{ OF} \\
& & \qquad \begin{array}{lcll}
* & \longmapsto & * & \text{// non-termination} \\
(s', y) & \longmapsto & p\,y\,s' & \text{// normal termination} \\
(s', e) & \longmapsto & (s', e) & \text{// except. termination}
\end{array}
\end{array}
$$

V. Java program verification (at Nijmegen)

Developments

Developments

- **Original focus:** theorem proving for small Java programs (for smart cards)

Developments

- **Original focus:** theorem proving for small Java programs (for smart cards)
- **Outcome:**
  - No scaling beyond couple of pages
  - Practical experience, formalisations & deeper theory

Developments

- **Original focus:** theorem proving for small Java programs (for smart cards)
- **Outcome:**
  - No scaling beyond couple of pages
  - Practical experience, formalisations & deeper theory
- **Shift of focus:**
  - Extension to security properties (esp. confidentiality)
  - Static checking primary, theorem proving secondary

JML: Java Modeling Language

JML: Java Modeling Language

JML [Leavens et al.] adds specifications as special comments in Java code, mainly for:

JML: Java Modeling Language

JML [Leavens et al.] adds specifications as special comments in Java code, mainly for:

- Class invariants and constraints

JML: Java Modeling Language

JML [Leavens et al.] adds specifications as special comments in Java code, mainly for:

- Class invariants and constraints
- Method specifications:

```
/*@ behavior
  @ requires   <precondition>
  @ assignable <items that may be modified>
  @ diverges   <precondition for non-termination>
  @ ensures    <postcond for normal termination>
  @ signals    <postcond for exceptional
  @             termination>
  @*/
void method() { ...  }
```

JML: example

JML: example

JML method specifications may clarify the behaviour of Java methods:

JML: example

JML method specifications may clarify the behaviour of Java methods:

```
int f(int x) {
   int count = 0, sum = 1;
   while (sum <= x) {
      count++;
      sum += 2 * count + 1;
   }
   return count;
}
```

JML: example

JML method specifications may clarify the behaviour of Java methods:

```
/*@ normal_behavior
  @ requires      x >= 0;
  @ assignable    \nothing;
  @ ensures       \result * \result <= x &&
  @               x < (\result+1) * (\result+1)
  @*/
int f(int x) {
  int count = 0, sum = 1;
  while (sum <= x) {
    count++;
    sum += 2 * count + 1;
  }
  return count;
}
```

LOOP project

LOOP project

- LOOP tool: compiles Java+JML to PVS

LOOP project

- LOOP tool: compiles Java+JML to PVS
- Based on formalised semantics of Java+JML in PVS

LOOP project

- LOOP tool: compiles Java+JML to PVS
- Based on formalised semantics of Java+JML in PVS
- Including Hoare logic (see later) &
  WP-reasoner
  (all with provably sound rules)

LOOP project

- LOOP tool: compiles Java+JML to PVS
- Based on formalised semantics of Java+JML in PVS
- Including Hoare logic (see later) & WP-reasoner
  (all with provably sound rules)
- Used for several non-trivial case studies, but now in “sleep mode”

LOOP project

- LOOP tool: compiles Java+JML to PVS
- Based on formalised semantics of Java+JML in PVS
- Including Hoare logic (see later) & WP-reasoner
  (all with provably sound rules)
- Used for several non-trivial case studies, but now in "sleep mode"
- Static checking is simply more effective; theorem proving best for difficult left-overs.

VI. Static Checking for Java

ESC/Java and ESC/Java2

ESC/Java and ESC/Java2

Extended static checker: original ESC/Java by Leino et. al at Compaq, but no longer supported.

ESC/Java and ESC/Java2

Extended static checker: original ESC/Java by Leino et. al at Compaq, but no longer supported.

- *tries* to *prove* correctness of specifications, at compile-time, fully automatically

ESC/Java and ESC/Java2

Extended static checker: original ESC/Java by Leino et. al at Compaq, but no longer supported.

- *tries* to *prove* correctness of specifications, at compile-time, fully automatically
- *not sound*, *not complete*, but finds lots of potential bugs quickly

ESC/Java and ESC/Java2

Extended static checker: original ESC/Java by Leino et. al at Compaq, but no longer supported.

- *tries* to *prove* correctness of specifications, at compile-time, fully automatically
- *not sound*, *not complete*, but finds lots of potential bugs quickly
- Original ESC/Java only supports a (not fully compatible) subset of full JML

ESC/Java and ESC/Java2

Extended static checker: original ESC/Java by Leino et. al at Compaq, but no longer supported.

- *tries* to *prove* correctness of specifications, at compile-time, fully automatically
- *not sound*, *not complete*, but finds lots of potential bugs quickly
- Original ESC/Java only supports a (not fully compatible) subset of full JML
- New ESC/Java2 is open source, compatible and handles more (eg. **`assignable`** clauses).

ESC/Java “demo”

```
class Bag {
  int[] a;
  int   n;
  int extractMin() {
   int m = Integer.MAX_VALUE;
   int mindex = 0;
   for (int i = 1; i <= n; i++) {
        if (a[i] < m) { mindex = i; m = a[i]; } }
   n--;
   a[mindex] = a[n];
   return m;
 }
```

ESC/Java “demo”

```
class Bag {
  int[] a;
  int   n;
  int extractMin() {
   int m = Integer.MAX_VALUE;
   int mindex = 0;
   for (int i = 1; i <= n; i++) {
        if (a[i] < m) { mindex = i; m = a[i]; } }
   n--;
   a[mindex] = a[n];
   return m;
 }
```

Warning: possible null deference. Plus other warnings

ESC/Java “demo”

```
class Bag {
  int[] a;  //@ invariant a != null;
  int   n;
  int extractMin() {
   int m = Integer.MAX_VALUE;
   int mindex = 0;
   for (int i = 1; i <= n; i++) {
        if (a[i] < m) { mindex = i; m = a[i]; } }
   n--;
   a[mindex] = a[n];
   return m;
 }
```

ESC/Java “demo”

```
class Bag {
  int[] a;  //@ invariant a != null;
  int   n;
  int extractMin() {
   int m = Integer.MAX_VALUE;
   int mindex = 0;
   for (int i = 1; i <= n; i++) {
        if (a[i] < m) { mindex = i; m = a[i]; } }
   n--;
   a[mindex] = a[n];
   return m;
 }
```

Warning: Array index possibly too large

ESC/Java “demo”

```
class Bag {
  int[] a;  //@ invariant a != null;
  int   n;  //@ invariant 0 <= n && n <= a.length;
  int extractMin() {
   int m = Integer.MAX_VALUE;
   int mindex = 0;
   for (int i = 1; i <= n; i++) {
        if (a[i] < m) { mindex = i; m = a[i]; } }
   n--;
   a[mindex] = a[n];
   return m;
 }
```

ESC/Java “demo”

```
class Bag {
  int[] a;  //@ invariant a != null;
  int   n;  //@ invariant 0 <= n && n <= a.length;
  int extractMin() {
   int m = Integer.MAX_VALUE;
   int mindex = 0;
   for (int i = 1; i <= n; i++) {
        if (a[i] < m) { mindex = i; m = a[i]; } }
   n--;
   a[mindex] = a[n];
   return m;
 }
```

Warning: Array index possibly too large

ESC/Java “demo”

```
class Bag {
  int[] a;  //@ invariant a != null;
  int   n;  //@ invariant 0 <= n && n <= a.length;
  int extractMin() {
   int m = Integer.MAX_VALUE;
   int mindex = 0;
   for (int i = 0; i < n; i++) {
        if (a[i] < m) { mindex = i; m = a[i]; } }
   n--;
   a[mindex] = a[n];
   return m;
 }
```

ESC/Java “demo”

```
class Bag {
  int[] a;  //@ invariant a != null;
  int   n;  //@ invariant 0 <= n && n <= a.length;
  int extractMin() {
   int m = Integer.MAX_VALUE;
   int mindex = 0;
   for (int i = 0; i < n; i++) {
        if (a[i] < m) { mindex = i; m = a[i]; } }
   n--;
   a[mindex] = a[n];
   return m;
 }
```

Warning: Possible negative array index

ESC/Java “demo”

```
class Bag {
  int[] a;  //@ invariant a != null;
  int   n;  //@ invariant 0 <= n && n <= a.length;
  //@ requires n > 0;
  int extractMin() {
   int m = Integer.MAX_VALUE;
   int mindex = 0;
   for (int i = 0; i < n; i++) {
        if (a[i] < m) { mindex = i; m = a[i]; } }
   n--;
   a[mindex] = a[n];
   return m;
 }
```

ESC/Java “demo”

```
class Bag {
  int[] a;  //@ invariant a != null;
  int   n;  //@ invariant 0 <= n && n <= a.length;
  //@ requires n > 0;
  int extractMin() {
   int m = Integer.MAX_VALUE;
   int mindex = 0;
   for (int i = 0; i < n; i++) {
        if (a[i] < m) { mindex = i; m = a[i]; } }
   n--;
   a[mindex] = a[n];
   return m;
 }
```

No more warnings about this code

ESC/Java “demo”

```
class Bag {
  int[] a;  //@ invariant a != null;
  int   n;  //@ invariant 0 <= n && n <= a.length;
  //@ requires n > 0;
  int extractMin() {
   int m = Integer.MAX_VALUE;
   int mindex = 0;
   for (int i = 0; i < n; i++) {
        if (a[i] < m) { mindex = i; m = a[i]; } }
   n--;
   a[mindex] = a[n];
   return m;
  }
```

. . . but warnings about calls to **`extractMin()`** that do not ensure precondition : design by contract

VII. Hoare logic for JML

Hoare logic issues for Java & JML

Hoare logic issues for Java & JML

- Complications in Hoare logic for Java:
  - exceptions and other abrupt control flow
  - expressions may have side effects

Hoare logic issues for Java & JML

- Complications in Hoare logic for Java:
  - exceptions and other abrupt control flow
  - expressions may have side effects
- Thus:
  - not Hoare *triples* but Hoare *n-tuples*,
  - both for statements & expressions

Hoare Logic assertions

Hoare Logic assertions

For $\{\,\mathit{Pre}\,\}\,m\,\{\,\mathit{Post}\,\}$ write

$$\begin{pmatrix} \text{requires} & = & \mathit{Pre} \\ \text{statement} & = & m \\ \text{ensures} & = & \mathit{Post} \end{pmatrix}$$

Hoare Logic assertions

For $\{\,\mathit{Pre}\,\}\, m \,\{\,\mathit{Post}\,\}$ write

$$\left(\begin{array}{rcl} \text{requires} & = & \mathit{Pre} \\ \text{statement} & = & m \\ \text{ensures} & = & \mathit{Post} \end{array}\right)$$

For JML one needs: $\left(\begin{array}{rcl} \text{diverges} & = & D \\ \text{requires} & = & \mathit{Pre} \\ \text{statement} & = & m \\ \text{ensures} & = & \mathit{Post} \\ \text{signals} & = & S \end{array}\right)$

Hoare composition Rule

Hoare composition Rule

$$\frac{\begin{pmatrix} \text{diverges} & = & \lambda x.\, b \\ \text{requires} & = & \mathit{Pre} \\ \text{statement} & = & s_1 \\ \text{ensures} & = & Q \\ \text{signals} & = & S \end{pmatrix} \qquad \begin{pmatrix} \text{diverges} & = & \lambda x.\, b \\ \text{requires} & = & Q \\ \text{statement} & = & s_2 \\ \text{ensures} & = & \mathit{Post} \\ \text{signals} & = & S \end{pmatrix}}{\begin{pmatrix} \text{diverges} & = & \lambda x.\, b \\ \text{requires} & = & \mathit{Pre} \\ \text{statement} & = & s_1\, ;\, s_2 \\ \text{ensures} & = & \mathit{Post} \\ \text{signals} & = & S \end{pmatrix}}$$

Hoare composition Rule

$$\dfrac{\begin{pmatrix} \text{diverges} & = & \lambda x.\, b \\ \text{requires} & = & \textit{Pre} \\ \text{statement} & = & s_1 \\ \text{ensures} & = & Q \\ \text{signals} & = & S \end{pmatrix} \qquad \begin{pmatrix} \text{diverges} & = & \lambda x.\, b \\ \text{requires} & = & Q \\ \text{statement} & = & s_2 \\ \text{ensures} & = & \textit{Post} \\ \text{signals} & = & S \end{pmatrix}}{\begin{pmatrix} \text{diverges} & = & \lambda x.\, b \\ \text{requires} & = & \textit{Pre} \\ \text{statement} & = & s_1 \,;\, s_2 \\ \text{ensures} & = & \textit{Post} \\ \text{signals} & = & S \end{pmatrix}}$$

Intermediate predicate provided by the user in JML

Use of the Hoare logic

Use of the Hoare logic

- Actual use seems clumsy, but PVS takes care of the bookkeeping

Use of the Hoare logic

- Actual use seems clumsy, but PVS takes care of the bookkeeping
- This logic forms basis for semantics of JML

VIII. Conclusions

Main points

Main points

- There is mathematical uniformity & elegance in the structure of computation

Main points

- There is mathematical uniformity & elegance in the structure of computation
- Main notions: monad / comonad / arrow

Main points

- There is mathematical uniformity & elegance in the structure of computation
- Main notions: monad / comonad / arrow
- This elegance is not completely lost in concrete languages / systems

Main points

- There is mathematical uniformity & elegance in the structure of computation
- Main notions: monad / comonad / arrow
- This elegance is not completely lost in concrete languages / systems
- For our Java work: practice preceded theory

Main points

- There is mathematical uniformity & elegance in the structure of computation
- Main notions: monad / comonad / arrow
- This elegance is not completely lost in concrete languages / systems
- For our Java work: practice preceded theory
- Theorem proving cannot beat static checking in program verification

Main points

- There is mathematical uniformity & elegance in the structure of computation
- Main notions: monad / comonad / arrow
- This elegance is not completely lost in concrete languages / systems
- For our Java work: practice preceded theory
- Theorem proving cannot beat static checking in program verification

Thanks for your attention!