I. Sneak preview

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
Writing $S$ for the type of states ...
... an imperative program is:

$$X \times S \rightarrow Y \times S$$

Or, equivalently,

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

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

Reactive, stream-based programs
A reactive program is:

$$X^N \rightarrow Y^N$$

Or, equivalently,

$$X^N \times N \rightarrow Y$$

Involving the Stream Comonad $X \mapsto X^N \times N$

Quantum program
A possible quantum program is:

$$X \times X \rightarrow \text{CmplxNr}((Y \times Y))$$

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
- Functional: $X \rightarrow Y$
- Imperative: $X \rightarrow T(Y)$, with $T$ monad (including Java programs)
- Reactive: $G(X) \rightarrow Y$, with $G$ comonad
- Quantum: $A(X, Y)$, with $A$ “arrow"
II. Comonads

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

- **Categorically:** endofunctor $G: C \to C$ with two natural transformations $\varepsilon: G \Rightarrow \text{Id}$ and $\delta: G \Rightarrow G^2$ satisfying standard equations
- **Computationally:** Type operator $G$ with
  - $\text{coreturn}: GX \to X$
  - $\text{cobind}: (GX \to Y) \to (GX \to GY)$ satisfying suitable equations
- **Logically:** structure for weakening and contraction (like bang ! in linear logic)

Comonad example

- Mapping $X \leftrightarrow X^N \times N$
- Input streams with past / current / future:
  $$x_0, x_1, \ldots, x_{n-1}, \underline{x_n}, x_{n+1}, x_{n+2}, \ldots$$
- Counit / coreturn: $X^N \times N \to X$
  $$(\alpha, n) \mapsto (\alpha(n))$$
- Delta: $X^N \times N \to (X^N \times N)^N \times N$
  $$(\alpha, n) \mapsto (\lambda m: N. (\alpha, m), n)$$
coKleisli category of computations

- coKleisli maps $X^N \times N \rightarrow Y$ 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

Three basic comonads:

\[
X^* \times X \quad \text{causality} \quad X^N \times N \quad \text{anti-causality} \quad X^N
\]

\[
((\alpha(0), \ldots, \alpha(n-1)), \alpha(n)) \quad \lambda m. \alpha(n+m)
\]

with “comonad homomorphisms” between them

Continuous time signals

Analogues fundamental diagram of comonads:

\[
\prod_{t \in [0, \infty)} X^{(0,t)} \times X \quad \text{causality} \quad \prod_{t \in [0, \infty)} X^{0, \infty} \times [0, \infty) \quad \text{anti-causality} \quad \prod_{t \in [0, \infty)} X^{0, \infty}
\]

where:

\[
\prod_{t \in [0, \infty)} X^{(0,t)} \times X \cong \prod_{t \in [0, \infty)} X^{0, t} \cong X^{[0,1]} \times [0, \infty)
\]

III. Arrows
**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`, `>>>`, `first`.
- Folklore claim: Arrows are Freyd categories (Power & Robinson’99)
- Recently substantiated by first describing arrows as **monoids** in a category of bifunctors $\mathbb{C}^{\text{op}} \times \mathbb{C} \to \text{Sets}$

**Arrow in Haskell**

Introduced as type class:

```haskell
class Arrow A where
  arr :: (X → Y) → A X Y
  (>>>) :: A X Y → A Y Z → A X Z
  first :: A X Y → A (X, Z) (Y, Z)
```

Which should satisfy 8 equations, such as:

- $(a >>> b) >>> c = a >>> (b >>> c)$
- $a >>> \text{arr}(1) = a$
- $\text{first}(\text{arr}(f)) = \text{arr}(f \times 1)$, etc

**Arrow examples**

- $(X, Y) \mapsto (X \to T(Y))$, for $T$ monad
  $(X, Y) \mapsto (G(X) \to Y)$, for $G$ comonad
- $(X, Y) \mapsto (X \times X \to \text{CmplxNr}^{(Y \times Y)})$ for quantum computation
- $(X, Y) \mapsto (X^N \to \mathcal{P}(Y^N))$ for “non-deterministic dataflow”
- $(X, Y) \mapsto (2 \times S^*) \times ((S^* \times X) \to (1 + (S^* \times Y)))$
  for Swierstra-Duponcheel parser that motivated Hughes

**Arrows, categorically**

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

  ![Diagram](A(X, Y) \xrightarrow{A(f, g)} A(X', Y'))

- $a \xrightarrow{\text{arr}(f) >>> a >>> \text{arr}(g)}$

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

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

$$M \otimes M \xrightarrow{\mu} M \otimes I \xrightarrow{e} M \cong I \otimes M \xrightarrow{\otimes 1} M \otimes M$$

satisfying the monoid equations
- A monoid in $\mathcal{C}^\op$ is precisely a monad!

Arrows are also monoids

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

Excurs: monads are monoids

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

$$F \otimes G = F \circ G \quad I = \text{Id}$$

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

$$M \otimes M \xrightarrow{\mu} M \xrightarrow{\eta} \text{Id}$$

satisfying the monoid equations
- A monoid in $\mathcal{C}^\mathcal{C}$ is precisely a monad!

Arrows, intuitively

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

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 (−)*
  - state (− × S)^S
  - non-determinism 𝒫 (powerset)
  - probability 𝒟 (distribution)

Java monad
- Definition [Jacobs & Poll’03]:
  \[ J(X) = (1 + S × X + S × E)^S \]
- 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 J is “argument evaluation, before use” (and not sequential composition ; )
- For \( a: X \rightarrow J(Y) \), and \( p: Y \rightarrow J(Z) \),

\[
p \bullet a = \lambda x: X. \lambda s: S. \\
\text{CASES } a \times s \text{ OF} \\
\qquad * \quad \mapsto \quad * \quad \quad // \text{non-termination} \\
\qquad (s', y) \quad \mapsto \quad p \ y \ s' \quad // \text{normal termination} \\
\qquad (s', e) \quad \mapsto \quad (s', e) \quad // \text{except. termination}
\]
V. Java program verification
(at Nijmegen)

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* [Leavens et al.] adds specifications as special comments in Java code, mainly for:
- Class invariants and constraints
- Method specifications:

```java
/*@ 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 method specifications may clarify the behaviour of Java methods:

```java
/*@ 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 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**

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).

```java
class Bag {
    int[] a;
    int n;
    int extractMin() {
        int m = Integer.MAX_VALUE;
        int minIndex = 0;
        for (int i = 1; i <= n; i++) {
            if (a[i] < m) {
                m = a[i];
                minIndex = i;
            }
        }
        n--;
        a[minIndex] = a[n];
        return m;
    }
}
```
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: possible null dereference. Plus other warnings

Warning: Array index possibly too large
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--; // Warning: Possible negative array index
            a[mindex] = a[n];
        }
        return m;
    }
}

// Warning: Array index possibly too large

Warning: Array index possibly too large
ESC/Java “demo”

```java
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”

```java
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

- 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

For \{ Pre \} m \{ Post \} write
\[
\begin{align*}
\text{diverges} &= D \\
\text{requires} &= Pre \\
\text{statement} &= m \\
\text{ensures} &= Post
\end{align*}
\]

For JML one needs:
\[
\begin{align*}
\text{diverges} &= \lambda x. b \\
\text{requires} &= Pre \\
\text{statement} &= s_1 \\
\text{ensures} &= Q \\
\text{signals} &= S \\
\text{ensures} &= Post \\
\text{signals} &= S
\end{align*}
\]

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

- 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!

Jacobs – Types’06, 18/4/06 – p.52/52