IFIP WG 2.8 meeting, April 2010

Side-Effect Localization for Lazy, Purely Functional Languages via *Aspects*

Kung Chen

National Cheng-chi University, Taiwan

Ongoing work, partial results published in PEPM'09

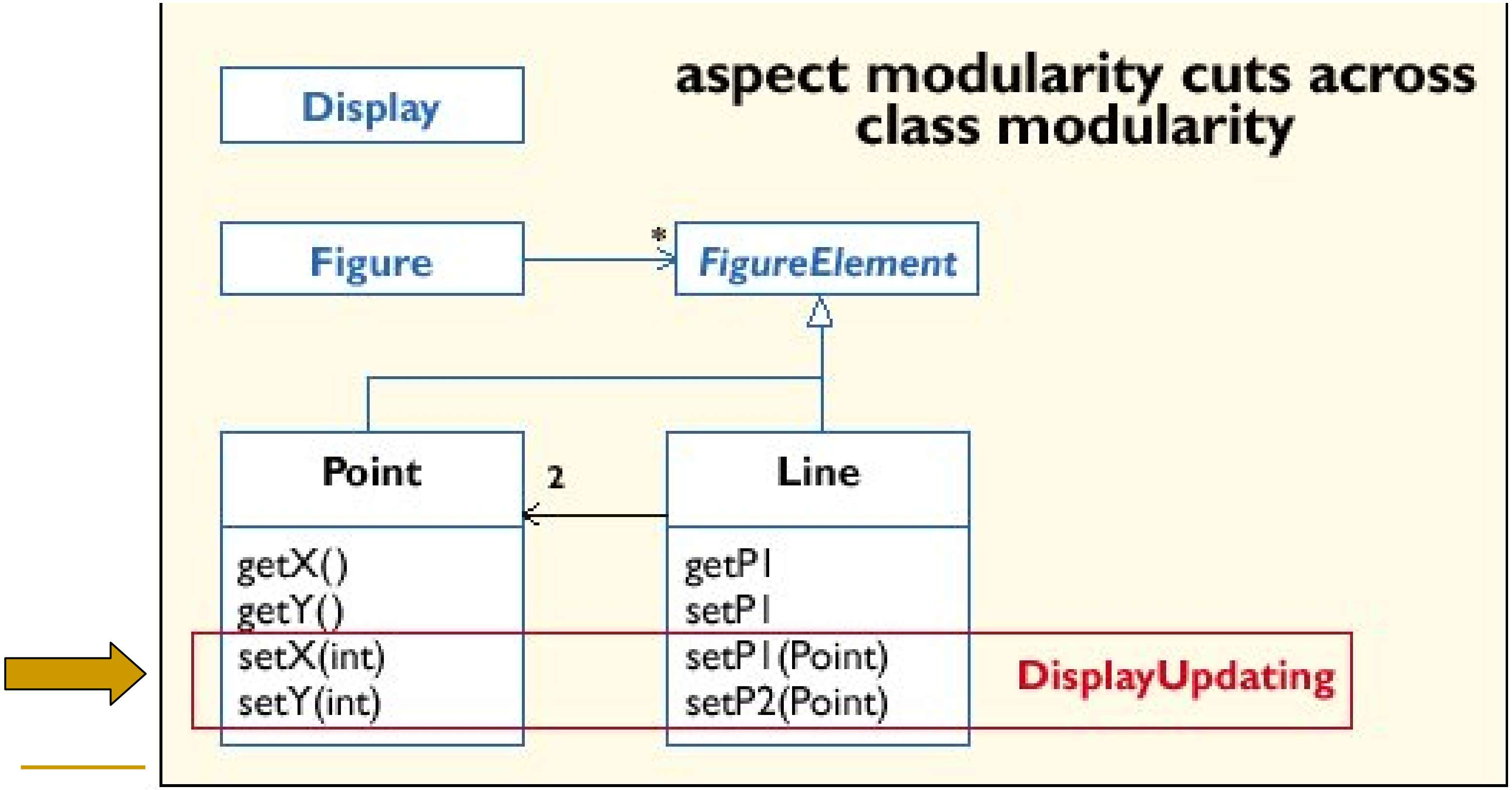
Join work with Shu-Chun Weng, Meng Wang and Siau-Cheng Khoo

Outline

- Introduction
  - AOP
  - Motivation
- AspectFun
  - Side-effecting aspects
- Transformations for Monad Introduction
- Issues & Extensions

Aspect-Oriented Programming (AOP)

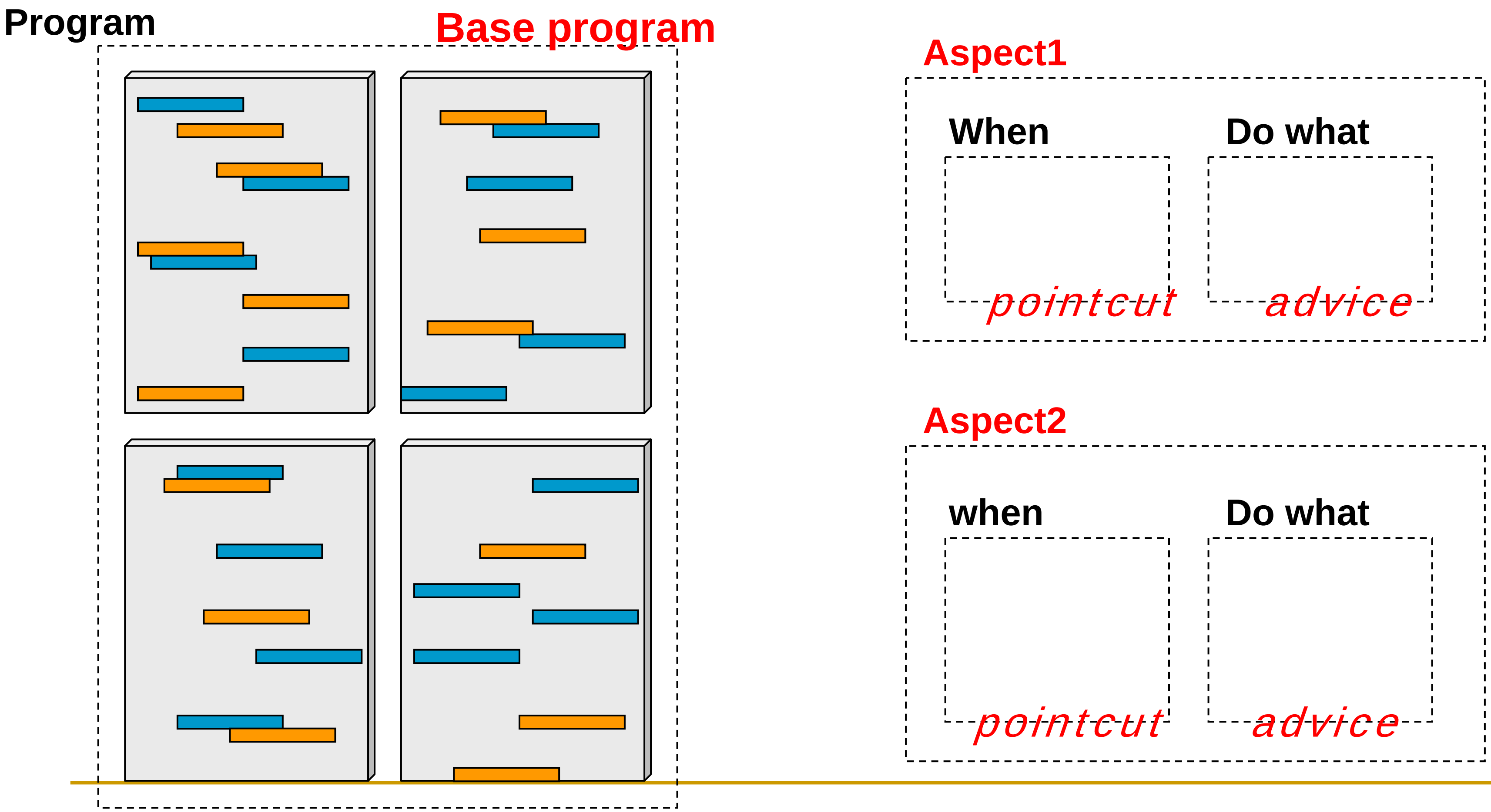
- **Aims at Improving modularity by addressing *crosscutting concerns;* hot in SE & OO communities.**
- **An example:**



Source: Communications of the ACM Vol. 44 No. 10, Pages 33-38

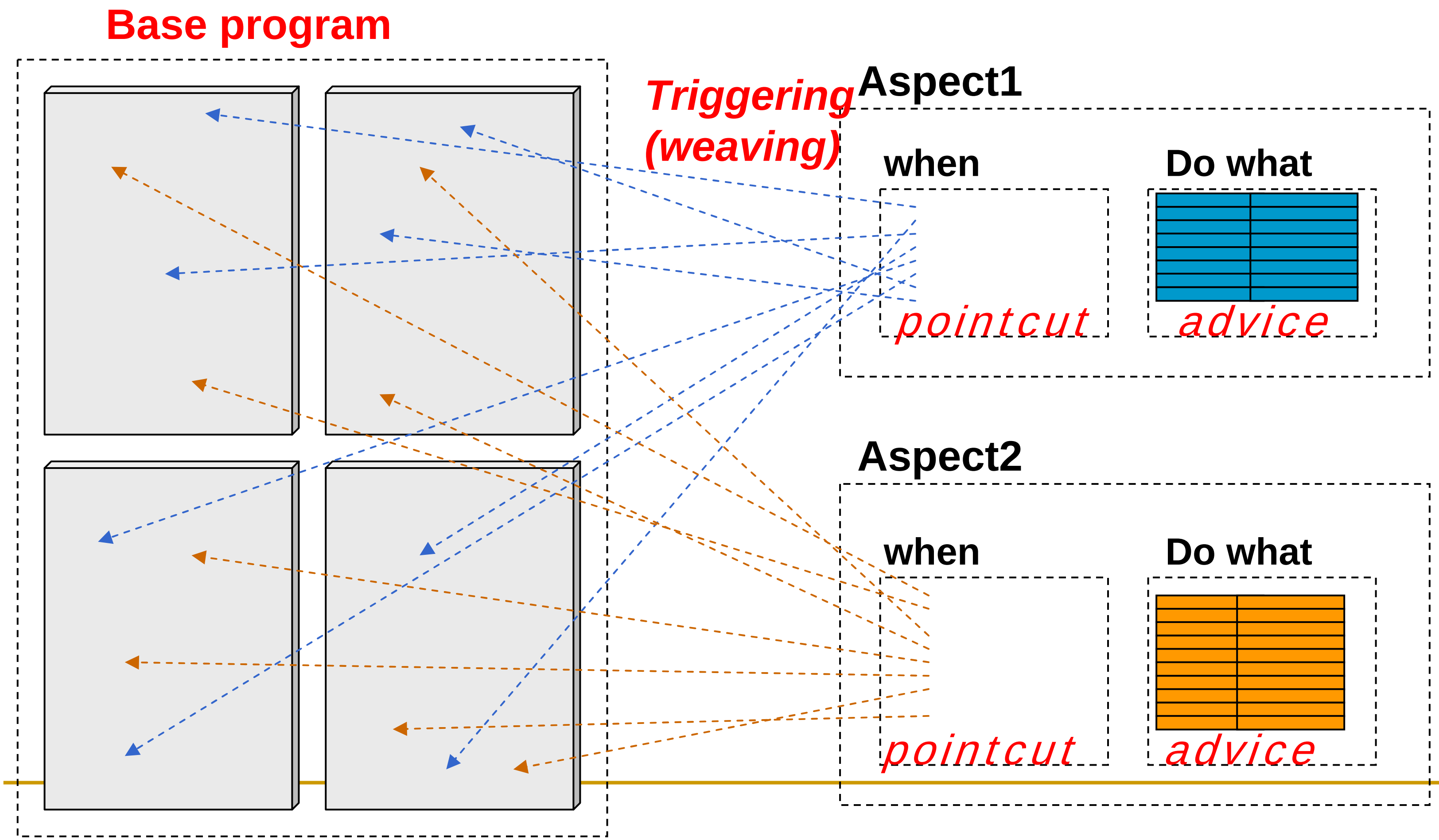
Aspect-Oriented Programming (AOP)

- **AOP programs are divided into *base programs* and *aspects***



Aspect-Oriented Programming (AOP)

- When the join points specified by the *pointcut* is reached during program execution, the advice in the aspects is triggered for execution.



Aspects

- Two parts of aspects
  - Pointcut: The specified points where intervention of execution take place
  - Advice: The action taken when the specified pointcut is reached.
    - Three kinds: before, after, around

Functional AOP

- **Applying AOP concepts to Functional Programming?**
  - Language implementations exist:
    - AspectML [Dantas et al, ICFP'05, TOPLAS'07],
    - Aspectual Caml [Masuhara et al, ICFP'05],
    - AspectFun [Chen et al, SAS'07, SCP'10]
      - Experimental, Haskell-like syntax
    - etc.

+Side-effecting aspects

Reference: an extensive survey of the impacts of AOP on (purely) functional programming:
What Does Aspect-Oriented Programming Mean for Functional Programmers?
M. Wang and B. Oliveira, WGP 2009

An Example: Memoization Aspect

```
fib 0 = 1
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

Base program

Aspect name

Pointcut

```
cache@advice around{fib} (arg) =
  if cacheContains(arg)
  then getCachedValue(arg)
  else setCache(arg, proceed(arg))
```

Aspect

Resume “fib”

Advice body

Static Weaving

The intervening action of aspects is realized by a **weaving** process.

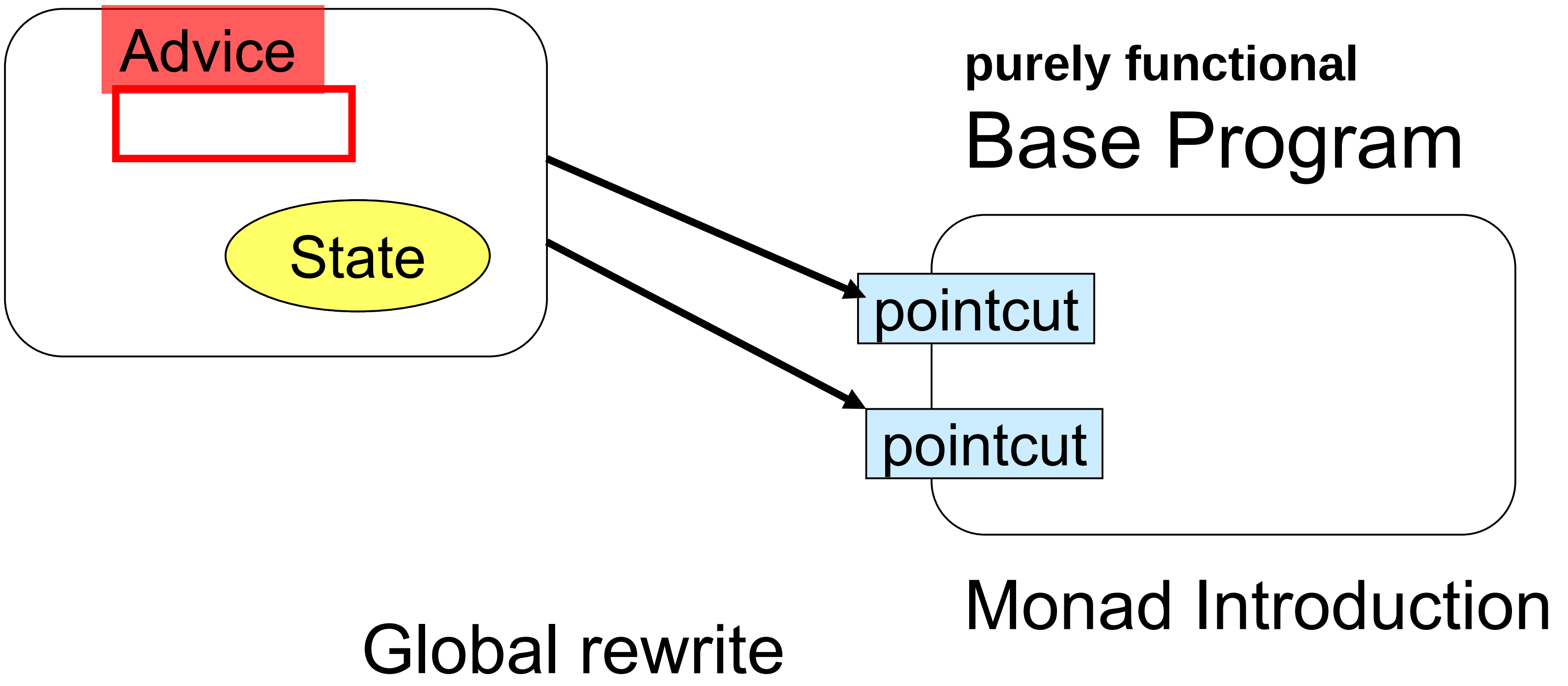
```
fib 0 = 1
fib 1 = 1
fib n = (cache fib) (n-1)
      + (cache fib) (n-2)

cache proceed arg =
  if cacheContains(arg)
  then getCachedValue(arg)
  else setCache(arg, proceed(arg))
```

Reference: [Chen et al, SAS'07, SCP'10]

Weaving Side-Effecting Aspects

Aspects with a state monad



Motivation

- Many useful aspects require *side-effecting computations,* but are “harmless.” [Dantas’06]
  - tracing, profiling, memoizaton, …
- Support side-effecting aspects directly on the language level and automate the rewriting using source-to-source transformations.

Related works: Monadification

- Automatic introduction of monads:
  - CPS conversions by Flanagan et al, and Hatcliff & Danvy
  - Monad introduction transformation by Lämmel
  - (Selective) Monadification by Erwig and Ren

•**But, as far as we know, no results for lazy, purely functional languages.**

Also related:
Purely Functional Lazy Non-deterministic Programming
by S. Fisher, O. Kiselyov, and C. C. Shan **in ICFP'09.**

Our Approach

- Linguistic support for side-effecting aspects

  by equipping AspectFun with
  - **Mutable variables**
  - **Output operation**

- A **type-directed monadification scheme** that transforms *woven code* into to a purely monadic style functional code using a **cache-enabled state monad.**

Outline

- Introduction
- AspectFun
  - Side-effecting aspects
- Transformations for monad introduction
  - The state monad
- Issues

AspectFun (Base programs)

- Haskell-like syntax
- Purely functional
- Polymorphic
- Lazy

```
fib n = if n <= 1 then 1
        else fib (n-1) + fib (n-2)

fac n acc = if n == 0 then acc
            else fac (n-1) (n*acc)
```

AspectFun (Base programs)

Programs	π	$::= d \text{ in } \pi \mid e$
Declarations	d	$::= x = e \mid f\ \overline{x} = e \mid f :: t \to t \mid$
		n@advice around $\{\overline{pc}\}\ (arg) = e$
Arguments	arg	$::= x \mid x :: t$
Pointcuts	pc	$::= ppc \mid pc + cf \mid pc - cf$
Primitive PC's	ppc	$::= f\ \overline{x} \mid \text{any} \mid \text{any} \backslash [\overline{f}] \mid n$
Cflows	cf	$::= \text{cflow}(f) \mid \text{cflow}(f(\_ :: t)) \mid$
		$\text{cflowbelow}(f) \mid \text{cflowbelow}(f(\_ :: t))$
Expressions	e	$::= c \mid x \mid \text{proceed} \mid \lambda x.e \mid e\ e \mid$
		$\text{if } e \text{ then } e \text{ else } e \mid \text{let } x = e \text{ in } e$
Types	t	$::= Int \mid Bool \mid a \mid t \to t \mid [t]$
Predicates	p	$::= (f : t)$
Advised Types	ρ	$::= p.\rho \mid t$
Type Schemes	σ	$::= \forall \bar{a}.\rho$

Reference: [Chen et al, SAS'07, SCP'10]

Side-Effecting Aspects: *mutable vars*

aspect name **where**
var id :: mono-type
advice around { pointcut } (arg) = exp

Example:

```
aspect memoFib where
  var memoMap:: Map.Map Int Int
  advice around {fib} (arg) =
      case lookupCache arg of
       Just v -> v
       Nothing ->
           set! v = proceed arg ;
           insertCache arg v;
           v
```

Two accessors:
-getMemoMap
-setMemoMap

--';' Sequencing

--set!:Sequenced bindings

Side-Effecting Aspects: *putMsg*

Example: a tracing aspect using set! and **putMsg**

```
fac n acc =
        if n == 0 then acc
        else fac (n - 1) (n * acc)
```

--putMsg *aString*

```
aspect tracer where
 var indent :: String = ""  --state
 advice around{ fac, (*) } (arg) = \arg2 ->
  set! ind = getIndent ;
  setIndent ("| " ++ ind);
  set! v1 = arg;
  set! v2 = arg2;
  putMsg  (ind++ tjp ++" receives ["++ show v1 ++ ", " ++ show v2 ++ "]");
  set! result = proceed v1 v2 ;
  setIndent ind;
  putMsg  (ind++tjp++" returns " ++  show result);
  result
```

show: Int->String

call-by-value tracing

tjp: this join point (function being advised)

Adapted from [Kishon 92]

Side-Effecting Aspects: *Tracing*

Example: *call-by-value* trace of “fac 3 1”

```
fac n acc = if n == 0 then acc
                       else fac (n - 1) (n * acc)
```

A lazy trace?
We’ll give one later.

fac 3 1 ➔

```
fac receives [3, 1]
| |  times receives [3, 1]
| |  times returns 3
|  fac receives [2, 3]
| |  | times receives [2, 3]
| |  | times returns 6
| |  fac receives [1, 6]
| |  |  | times receives [1, 6]
| |  |  | times returns 6
| |  |  fac receives [0, 6]
| |  |  fac returns 6
| |  fac returns 6
|  fac returns 6
fac returns 6
```

Outline

- Introduction
- AspectFun
  - Side-effecting aspects
- Transformations for monad introduction
  - The state monad
- Issues
  - Lazy evaluation
  - Higher-Order functins

Transformations for monad introduction

1. Apply A-normalization to woven code
2. Perform Type-directed monadification
   - Monadify woven code w.r.t **(*m*, *return*, >>=)** in a type context Γ
     - **$[\![e]\!]_{\Gamma} = e_M$**
3. Specialized ***m*** to a *state monad with a cache facility*

1) A-normalization: every intermediate computation is assigned a name by a LET expression; applications become (e a) – a: an atomic exp (var or const)

```
fac n acc =                --woven code
  if n == 0 then acc
  else (tracer fac) (n-1) ((tracer (mul)) n acc)
```

```
facA :: Int -> Int -> Int
facA n acc =
    let nleq0 = n == 0 in
      if nleq0 then acc
       else let nmacc = (tracer (mul)) n acc in
             let nm1 = n – 1 in
               (tracer facA) nm1 nmacc
```

2) Monadification (later)

3) The State Monad

- Concrete monad to support side-effecting aspects:
- ```
  data State s a = State
        { runState :: s -> (a, s) }
  ```

- ```
  type M a =
       State (UserVar, OutBuf) a
  ```

A record of mutable variables

Output buffer for putMsg

(2)Type-directed monadification

- Rewriting function $[| \; |]_\Gamma$ **:: exp → exp^m** that lifts computations in the input expression to a designated monad of *(m, return, >>).*

- The def of [| |] is guided by the *monadic types* assigned by the following *monadification operator*, $\mathcal{M}$:: Type → Type

Type t ::= Int | Bool | a | t -> t

- Goal: If $\Gamma \vdash e : t$,
  then $\mathcal{M}(\Gamma) \vdash [|e|]_\Gamma : \mathcal{M}(t)$

Ex: fac :: Int->Int->Int , $[|\; fac \;|]_\Gamma :: \mathcal{M}(Int\text{->}Int\text{->}Int)$

(2)Type-directed monadification

The def of [| |] is guided by the *monadic types* assigned by the following *monadification operator*,

Type t ::= Int | Bool | a | t -> t

$\mathcal{M}$:: Type → Type

- $\mathcal{M}(t_1 \to t_2) \Rightarrow \mathcal{M}(t_1) \to \mathcal{M}(t_2)$
- $\mathcal{M}(t) \Rightarrow m\ t$ ***if t is non-functional (atomic)***
- $\mathcal{M}(\forall \overline{a}.t) \Rightarrow \forall \overline{a}.\mathcal{M}(t)$

Ex: fac :: Int->Int->Int , $\mathcal{M}$(Int->Int->Int) = m Int -> m Int -> m Int

Monadification (First try)

- $[|e|]_\Gamma$ lifts expressions to monadic space

	$[\![\cdot]\!]_\Gamma$	:	$e \longrightarrow e^M$
(CONST)	$[\![c]\!]_\Gamma$	=	return c
(PRIM)	$[\![p]\!]_\Gamma$	=	`liftMn` p *where* n is the arity of primitive function p
(VAR)	$[\![x]\!]_\Gamma$	=	x
(IF)	$[\![$if a then e_1 else $e_2]\!]_\Gamma$	=	*let* $e_1^M = [\![e_1]\!]_\Gamma$ $e_2^M = [\![e_2]\!]_\Gamma$ *in if* $isConst(a)$ *then* if a then e_1^M else e_2^M *else* do $\{x' \leftarrow$ [|a|]; if x' then e_1^M else $e_2^M\}$ *where* x' is fresh
(LAM)	$[\![\lambda x.e]\!]_\Gamma$	=	$\lambda x.[\![e]\!]_\Gamma$
(APP)	$[\![e\ a]\!]_\Gamma$	=	$[\![e]\!]_\Gamma\ [\![a]\!]_\Gamma$
(LET)	$[\![$let $x = e_1$ in $e_2]\!]_\Gamma$	=	*let* $e_1^M = [\![e_1]\!]_\Gamma$ $e_2^M = [\![e_2]\!]_\Gamma$ *in* do $\{$let $x = e_1^M;\ e_2^M\}$
	where $a \in$ Atoms	::=	$c \mid x$

Sequencings and set!

$$[\![e_1;\ e_2]\!]_\Gamma \;=\; \mathsf{do}\ \{[\![e_1]\!]_\Gamma\ ;\ [\![e_2]\!]_\Gamma\}$$

[| set! x = e1 ; e2 |] =

$$\begin{aligned} &let\ e_1^M = [\![e_1]\!]_\Gamma \\ &\qquad e_2^M = [\![e_2]\!]_\Gamma \\ &in\ \ \mathsf{do}\ \{x' \leftarrow e_1^M;\ \mathsf{let}\ x = \mathsf{return}\ x';\ e_2^M\} \\ &where\ x'\ \text{is a fresh identifier} \end{aligned}$$

The Monadification Operator

- Previous works:

  $$\mathcal{M}(t_1 \rightarrow t_2) \Rightarrow t_1 \rightarrow \mathcal{M}(t_2)$$

- Ours (Call-By-Name)

  $$\mathcal{M}(t_1 \rightarrow t_2) \Rightarrow \mathcal{M}(t_1) \rightarrow \mathcal{M}(t_2)$$

  - ***(m a)*** **is an** action **as well as a** thunk

- An alternative:
  - $\mathcal{M}(t1 \rightarrow t2) \Rightarrow \mathrm{m}\ (\mathcal{M}(t1) \rightarrow \mathcal{M}(t2))$

2) Monadification w.r.t (m, return, >>=)

```
facA :: Int -> Int -> Int
facA n acc =
    let nleq0 = n == 0 in
      if nleq0 then acc
       else let nmacc = (tracer (mul)) n acc in
             let nm1 = n – 1 in
               (tracer facA) nm1 nmacc
```

```
facM :: m Int -> m Int -> m Int
facM n acc =
 do let n_eq_0 = (liftM2 (==)) n (return 0)
    neq0 <- n_eq_0
    if neq0 then acc
    else
      do let nmacc = (tracerM (liftM2 (mul)) n acc
         let nm1 = (liftM2 (-)) n (return 1)
         (tracerM facM) nm1 nmacc
```

Outline

- Introduction
- AspectFun
  - Side-effecting aspects
- Transformations for monad introduction
  - The state monad
- Issues
  - Lazy evaluation
  - Higher-Order functins

Issues of the Monadification Scheme

- **Preserving lazy evaluation order**
- **Higher-Order functions**
- **…**

A Lazy Tracer?

- Revise the call-by-value tracing aspect:

```
aspect tracer where
  var indent :: String = "“ --state
  advice around{ fac, (*) } (arg) = \arg2 ->
    set! ind = getIndent ;
    setIndent ("| " ++ ind);
    putMsg  (ind++ tjp ++" receives ["++ show arg ++ ", " ++ show arg2 ++ "]");
    set! result = proceed arg arg2 ;
    setIndent ind;
    putMsg  (ind++tjp++" returns " ++  show result);
    result
```

Monadified code (Not so “Lazy” tracer)

```
tracer@advice around{fac, (*)} (arg) =
 \arg2 -> let! ind = getIndent in
   setIndent ("| " ++ ind);
   putMsg (ind++tjp++" receives ["++show arg++", "++show arg2++"]");
   set! result = proceed arg arg2;
   setIndent ind;
   putMsg (ind ++ tjp ++ " returns " ++ show result);
   result
```

```
tracerM  proceed arg arg2 =
 do getIndentResult  <- getIndentM
    let ind = return getIndentResult
    let ind' = (liftM2 (++)) (return "| ") ind
    setIndentM ind'
    let show_arg2 = (liftM show) arg2
    let str_1 = (liftM2 (++)) show_arg2 (return "]")
    let str_2 = (liftM2 (++)) (return ",") str_1
    let show_arg = (liftM show) arg
    …
    putMsgM str_5
    proceedResult  <- proceed arg arg2
    …
```

Expected trace result

```
fac n acc = if n == 0 then acc
            else fac (n-1) (n*acc)
```

```
fac receives [3, 1]
| fac receives [2, 3]
| | fac receives [1, 6]
| | | fac receives [0, 6]
| | | | (*) receives [1, 6]
| | | | | (*) receives [2, 3]
| | | | | | (*) receives [3, 1]
| | | | | | (*) returns 3
| | | | | (*) returns 6
| | | | (*) returns 6
| | | fac returns 6
| | fac returns 6
| fac returns 6
fac returns 6
```

(*) are all done at the end according to lazy semantics

Actual trace result: duplicate evaluation and wrong order

accumulating parameter

```
fac receives [3, 1]
| | (*) receives [3, 1]
| | (*) returns 3
| fac receives [2, 3]
| | | | (*) receives [3, 1]
| | | | (*) returns 3
| | | (*) receives [2, 3]
| | | | (*) receives [3, 1]
| | | | (*) returns 3
| | | (*) returns 6
| | fac receives [1, 6]
: :
| | | | | | (*) receives [3, 1]
| | | | | | (*) returns 3
| | | | | (*) returns 6
| | | | (*) returns 6
| | | fac returns 6
| | fac returns 6
| fac returns 6
fac returns 6
```

Showing *arg2* forces the *multiplication* being called (Premature evaluation)

It is evaluated every time whenever (3*1) is needed (Duplicate evaluation)

```
fac n acc = if n == 0 then acc
            else fac (n-1) (n*acc)
```

Our Solution, 1

- Wrap the *State monad* with a cache facility to support lazy evaluation: CState monad
  - Insert “add2cache” for thunkifying function arguments

```
facM n acc =
  do eq_n_zero <- add2Cache $ (liftM2 (==)) n (return 0)
     neq0 <- eq_n_zero
     if neq0 then acc
     else do nmacc <-
               add2Cache $ (tracerMulM (liftM2 (*)) n acc
             nm1 <-
               add2Cache $ (liftM2 (-)) n (return 1)
             (tracerFacM facM) nm1 nmacc
```

CState: Cache-Enabled State Monad

```
-- Cells: thunks or values
data Cell = forall s a. Cell Bool (CState s a)
type Cache = Map.Map Int (Maybe Cell)

newtype CState s a = CState{ realrunCState ::
                  (s, Cache) -> (Either a Int, (s, Cache))}

type M a = CState (UserVar, OutputBuf) a
```

Our Solution, 2

- Provide a special `showM` to replace ordinary `show` function
  - `showM :: m Int -> m String`
  - return the thunk cell without forccing its evaluation
- Post-process the output

```
tracerFacM  proceed arg arg2 =
  do getIndentResult  <- getIndentM
     let ind = return getIndentResult
     let ind' = (liftM2 (++)) (return "| ") ind
     setIndentM ind'
     let show_arg2 = showM arg2
     ...
```

Issues with Higher-Order Functions

- The monadification, [| e |], does not work for higher-order functions.
  - The case of "(Var) [| x |] = x" is to blame.
- Example: [| (id_1 id_2) |]

[| (id_1 id_2) |] = [| id_1 |] [| id_2 |] = (~~id_1~~ id_2)

Because this does not type check!

S=[I->I/a]

[| id |] :: m a -> m a

If we specialize id_2 to Int->Int:

id_1 :: (I->I)->(I->I)

[| id_1 |] :: $\boxed{\text{m (m I -> m I)}}$ ->m (m I -> m I)

id_2 :: (I->I)

[| id_2 |] :: $\boxed{\text{(m I -> m I)}}$

$\mathcal{M}$ and S do Not distribute with each other

S: type substitution
$\mathcal{M}$(S): Monadified sub

$$\mathcal{M}(S)(a) = \begin{cases} Sa & \text{if } Sa \text{ is an atomic type} \\ \mathcal{M}(Sa) & \text{otherwise} \end{cases}$$

$$\begin{tikzcd}
t \arrow[r] \arrow[d, "S"'] & \mathcal{M}(t) \arrow[d, "\mathcal{M}(S)"] \\
St \arrow[r] & \mathcal{M}(St)
\end{tikzcd}$$

$\mathcal{M}$(St) != $\mathcal{M}$(S)$\mathcal{M}$(t)

Need Boilerplate Code to Make it Work

$$
\begin{tikzcd}
t \arrow[rr, "\mathcal{M}(\cdot)"] \arrow[dd, "S"'] & & \mathcal{M}(t) \arrow[d, "\mathcal{M}(S)"] \\
& & \mathcal{M}(S)\mathcal{M}(t) \arrow[dl, bend left, "\mathtt{pos}_t^S"] \\
St \arrow[r, "\mathcal{M}(\cdot)"] & \mathcal{M}(St) \arrow[ur, bend left, "\mathtt{neg}_t^S"] &
\end{tikzcd}
$$

[| (id_1 id_2) |] $^{(I\text{->}I)}$

= [| id_1|] $^{(I\text{->}I)\text{->}(I\text{->}I)}$ id_2

$$
\begin{aligned}
& [\![\mathrm{id}_1]\!]_\Gamma^{(Int \to Int) \to (Int \to Int)} \\
= \ & \mathtt{pos}^S_{a \to a}(\mathrm{id}_1) \\
= \ & \lambda x.\ \mathtt{pos}^S_a(\mathrm{id}_1\ \mathtt{neg}^S_a(x)) \\
= \ & \lambda x.\ \mathtt{flatten}[Sa]\ (\mathrm{id}_1\ (\mathsf{return}\ x)) \\
= \ & \lambda x.\ \mathrm{flatten}\ (\mathrm{id}_1\ (\mathsf{return}\ x)) \\
& \text{where } S = [a \mapsto (Int \to Int)] \\
& \qquad \mathrm{flatten} = \lambda v. \lambda x.\ v \gg\!= \lambda v'.\ v'\ x
\end{aligned}
$$

Type-Directed Monadification (Second try)

$[\![e]\!]^{t}_{\Gamma}$ Note: t is the type of e

$$
\begin{array}{lll}
 & [\![\cdot]\!]^t_\Gamma & : \; e^! \longrightarrow e \\
(\textsc{Const}) & [\![c]\!]^t_\Gamma & = \mathsf{return}\ c \\
(\textsc{Prim}) & [\![p]\!]^t_\Gamma & = \mathtt{liftMn}\ p \quad \textit{where}\ \mathtt{n}\ \text{is the arity of primitive function}\ p \\
(\textsc{If-C}) & [\![\mathsf{if}\ a\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2]\!]^t_\Gamma & = \mathsf{if}\ a\ \mathsf{then}\ [\![e_1]\!]^t_\Gamma\ \mathsf{else}\ [\![e_2]\!]^t_\Gamma \quad \textit{if}\ a\ \text{is constant} \\
(\textsc{If}) & [\![\mathsf{if}\ a\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2]\!]^t_\Gamma & = [\![a]\!]^{Bool}_\Gamma \gg\!= \lambda a'.\mathsf{if}\ a'\ \mathsf{then}\ [\![e_1]\!]^t_\Gamma\ \mathsf{else}\ [\![e_2]\!]^t_\Gamma \quad \textit{otherwise},\ a'\ \text{is fresh} \\
(\textsc{Lam}) & [\![\lambda x.e]\!]^{t_1 \to t_2}_\Gamma & = \lambda x.[\![e]\!]^{t_2}_\Gamma \\
(\textsc{App}) & [\![e\ a]\!]^t_\Gamma & = [\![e]\!]^{t_a \to t}_\Gamma\ [\![a]\!]^{t_a}_\Gamma \\
(\textsc{Let}) & [\![\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2]\!]^t_\Gamma & = \mathsf{let}\ x = [\![e_1]\!]^{t_x}_\Gamma\ \mathsf{in}\ [\![e_2]\!]^t_\Gamma \\
 & & \\
(\textsc{Seq}) & [\![e_1;\ e_2]\!]^t_\Gamma & = [\![e_1]\!]^{t_1}_\Gamma \gg\!= \lambda\_.[\![e_2]\!]^t_\Gamma \\
(\textsc{Set}) & [\![\mathsf{set!}\ x = e_1;\ e_2]\!]^t_\Gamma & = [\![e_1]\!]^{t_1}_\Gamma \gg\!= \lambda x'.\mathsf{let}\ x = \mathsf{return}\ x'\ \mathsf{in}\ [\![e_2]\!]^t_\Gamma \\
(\textsc{Var}) & [\![x]\!]^t_\Gamma & = \mathsf{pos}^S_{t'}(x) \\
 & & \quad \text{where}\ \forall\bar{a}.t' = \Gamma(x)\ \text{and}\ S\ \text{is a substitution such that}\ t = St'
\end{array}
$$

The Var case:

join :: m (m a) -> m a

$$[\!|\, x \,|\!]^t_\Gamma \;=\; \mathbf{pos}^S_{t'}(x)$$

where $\forall \bar{a}.t' = \Gamma(x)$ and S is a substitution such that $t = St'$

$$
\begin{array}{lll}
\mathtt{pos}^S_{t_1 \to t_2}(e) & = \lambda x.\ \mathtt{pos}^S_{t_2}(e\ \mathtt{neg}^S_{t_1}(x)) & x \notin fv(e) \\
\mathtt{pos}^S_a(e) & = \mathtt{flatten}[Sa]\ (e) & \text{if } a \in \mathrm{dom}(S) \\
\mathtt{pos}^S_t(e) & = e & \text{otherwise}
\end{array}
$$

$$
\begin{array}{lll}
\mathtt{neg}^S_{t_1 \to t_2}(e) & = \lambda x.\ \mathtt{neg}^S_{t_2}(e\ \mathtt{pos}^S_{t_1}(x)) & x \notin fv(e) \\
\mathtt{neg}^S_a(e) & = \mathsf{return}\ (e) & \text{if } a \in \mathrm{dom}(S) \text{ and } Sa \text{ is not an atomic type} \\
\mathtt{neg}^S_t(e) & = e & \text{otherwise}
\end{array}
$$

$$
\begin{array}{lll}
\mathtt{flatten}[t_1 \to t_2 \cdots t_n \to t](e) & = (\lambda x_1 \cdots x_n.\ e \gg\!= \lambda e'.\ e'\ x_1 \cdots x_n) & \text{where } n \text{ is the arity of the} \\
& & \text{functional type, } t \text{ is atomic} \\
\mathtt{flatten}[t](e) & = e & \text{for atomic type } t
\end{array}
$$

[type indexed]

Type Correctness

- Theorem: If $\Gamma \vdash e : t$,

  then $\mathcal{M}(\Gamma) \vdash [|e|]^{t}_{\Gamma} : \mathcal{M}(t)$

•Dynamic Semantics and Value Preservation issue is not covered in this talk.

More Issues

- Lifting higher-order primitives, such as '$'
  - Related: library functions without source code
  - Example:

```
            liftM2 ($) :: m (a->b)      -> m a -> m b
But we need         (m a -> m b) -> m a -> m b
```

  - A type-indexed "liftM[…](e)" ?
- Monadifying constructed data types such as lists
  - m [a] vs.
  - MList a = MNil | MCons (m a) (m (Mlist a))
- …

Extension: Monadic base programs

Use monad transformers:

- $\mathcal{M}(t_1 \to t_2) \Rightarrow \mathcal{M}(t_1) \to \mathcal{M}(t_2)$
- $\mathcal{M}(a) \Rightarrow MT\ N\ a$
- $\mathcal{M}(N\ (t_1 \to t_2)) \Rightarrow MT\ N\ (\mathcal{M}(t_1) \to \mathcal{M}(t_2))$
- $\mathcal{M}(N\ a) \Rightarrow MT\ N\ a$

- ```
  type CStateT s m a =
             CacheT (StateT s m) a
  ```
  ```

Summary

- Extending AspectFun with side-effecting aspects
- Type-directed monadification scheme
- Working on issues with higher-order functions

Thank you for listening.