Algebraic effects for idioms and arrows

Sam Lindley

Mathematically Structured Programming Group
University of Strathclyde

sam.lindley@strath.ac.uk

April 22, 2013

Idea

- Algebraic effects describe *abstract computations*
- Abstract computations are *trees* (modulo equations)
- Unrestricted computation trees are abstract *monadic* computations
- Computation trees with static *control flow* are abstract *arrow* computations
- Computation trees with static *control flow* and static *data flow* are abstract *idiom* (applicative functor) computations
- We can use special *flow* effects to enforce these restrictions, giving rise to a uniform treatment of monad, arrow and idiom computation

Talk plan

- I'll show you lots of pictures of computation trees

Talk plan

- I'll show you lots of pictures of computation trees
- If we haven't decided its time to go to the pub, I'll also present a type system

Abstract computations as trees

Given

- a signature of operations $E = \{\mathrm{op}_i : A_i \to B_i\}_i$
- a return type A

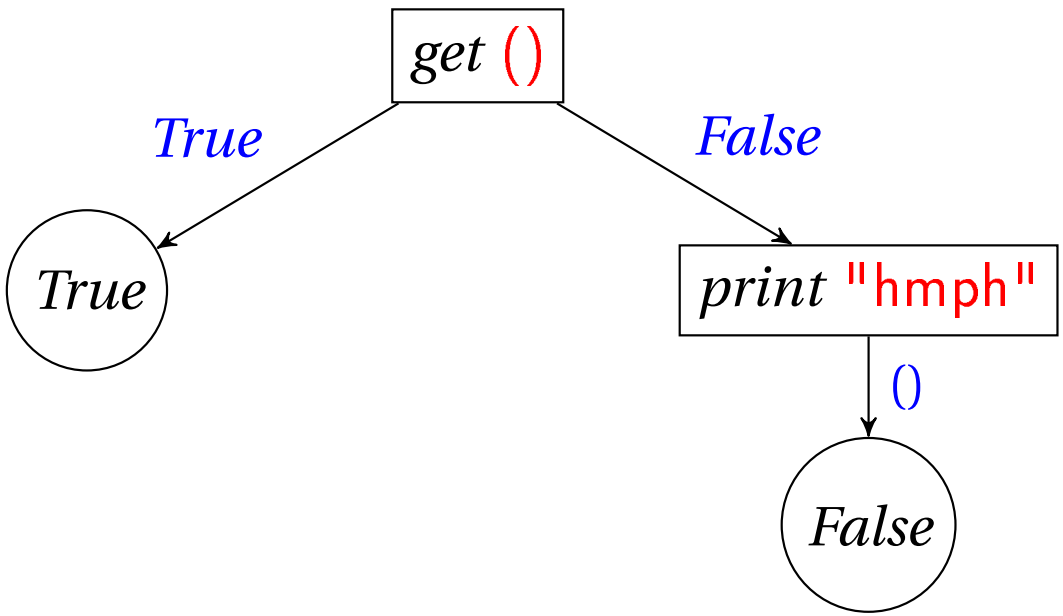
an abstract computation of type $\{[A]\}_E$ is a tree where

- nodes are labelled with operations and operation parameters
- edges are labelled with operation result values
- leaves are labelled with final return values of type A

Example 1 (hmph)

$$A = Bool$$
$$E = \{get \quad : () \quad \rightarrow Bool$$
$$\quad print : String \rightarrow ()\}$$

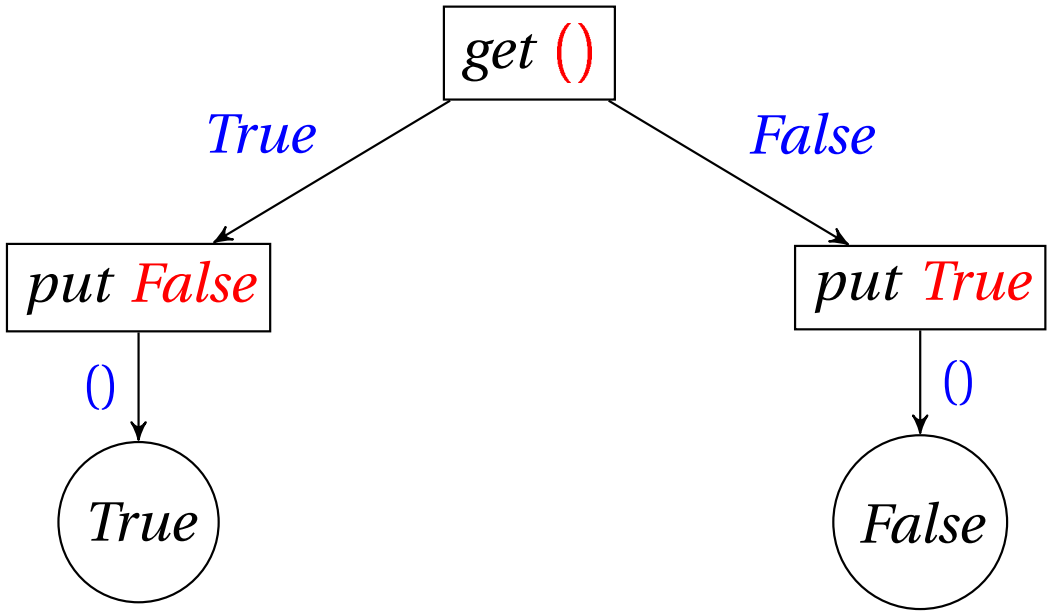
$$example1 = \textbf{do}\ x \leftarrow get\ (); \textbf{if}\ x\ \textbf{then}\ return\ x$$
$$\textbf{else}\ print\ \texttt{"hmph"}; return\ x$$



Example 2 (flip)

$$A = Bool$$
$$E = \{get : () \rightarrow Bool$$
$$\quad put : Bool \rightarrow ()\}$$

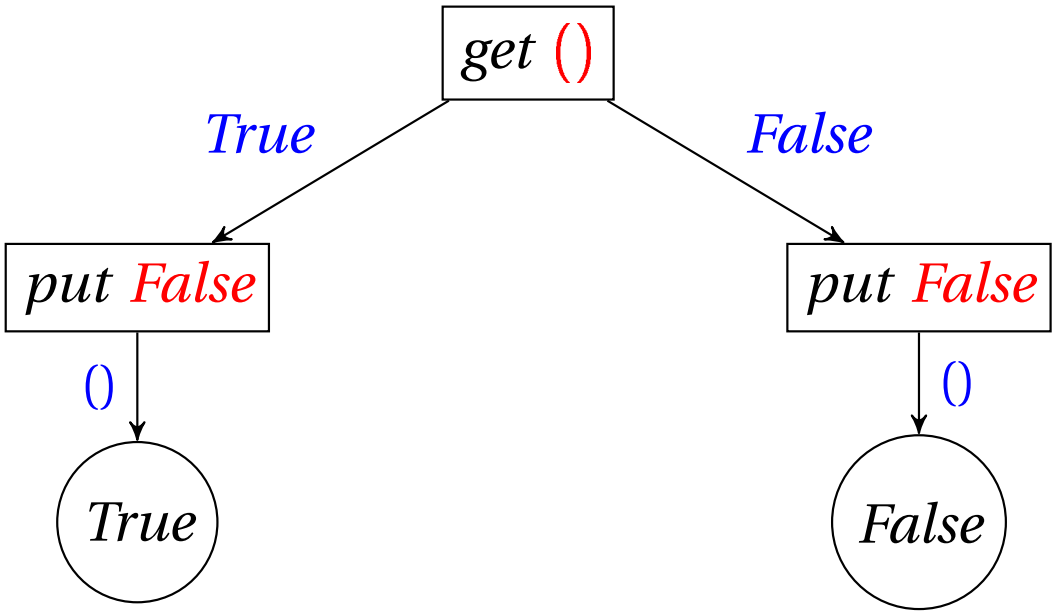
$$example2 = \mathbf{do}\ x \leftarrow get\ (); put\ (\neg\ x); return\ x$$



Example 3 (reset)

$$A = Bool$$
$$E = \{get : () \rightarrow Bool$$
$$\quad put : Bool \rightarrow ()\}$$

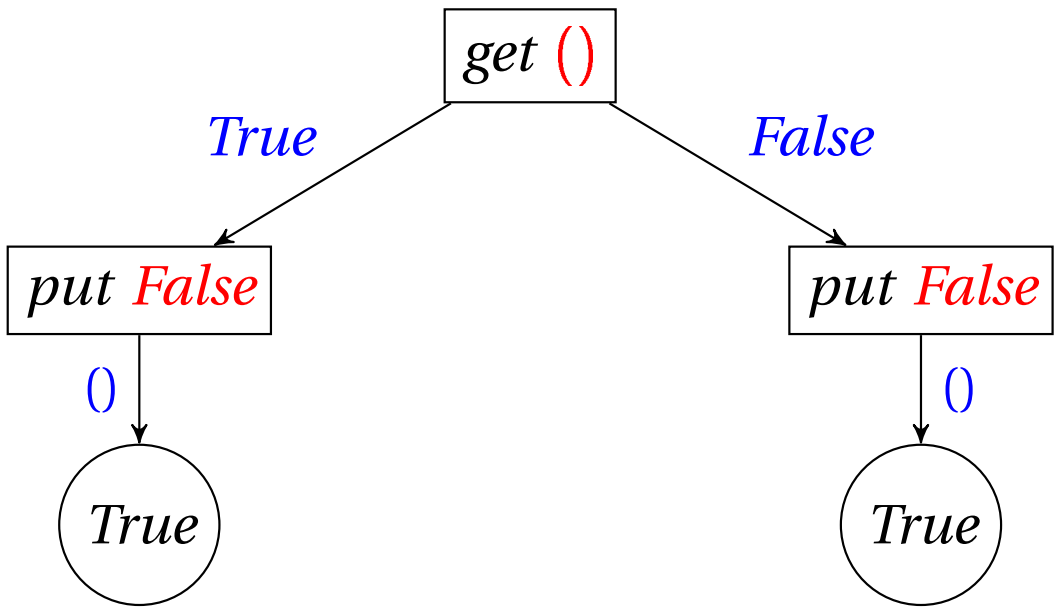
$$example3 = \textbf{do}\ x \leftarrow get\ (); put\ False; return\ x$$



Example 4 (reset and forget)

$$A = Bool$$
$$E = \{get : () \to Bool$$
$$\quad put : Bool \to ()\}$$

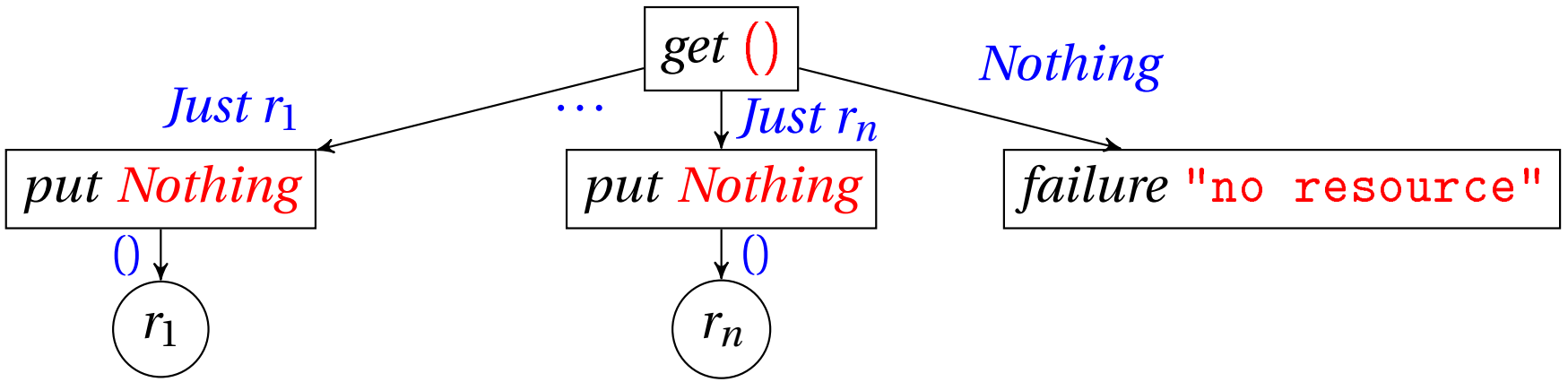
$$example4 = \textbf{do}\ x \leftarrow get\ (); put\ False; return\ True$$



Example 5 (acquire)

$$
\begin{aligned}
A &= \mathit{Resource} \\
E &= \{\mathit{get} &&: () &&\to \mathit{Maybe\ Resource} \\
&\quad\ \mathit{put} &&: \mathit{Maybe\ Resource} &&\to () \\
&\quad\ \mathit{failure} &&: \mathit{String} &&\to \mathit{Empty}\}
\end{aligned}
$$

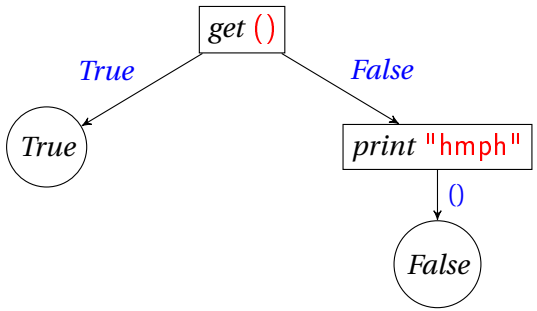
$$
\begin{aligned}
\mathit{example5} = \mathbf{do}\ & x \leftarrow \mathit{get}\ (); \\
& \mathbf{case}\ x\ \mathbf{of} \\
& \quad \mathit{Just}\ r \to \mathbf{do}\ \{\mathit{put}\ \mathit{Nothing}; \mathit{return}\ r\} \\
& \quad \mathit{Nothing} \to \mathbf{do}\ \{x \leftarrow \mathit{failure}\ \texttt{"no resource"} \\
& \qquad\qquad\qquad \mathbf{case}\ x\ \mathbf{of}\ \{\}\}
\end{aligned}
$$



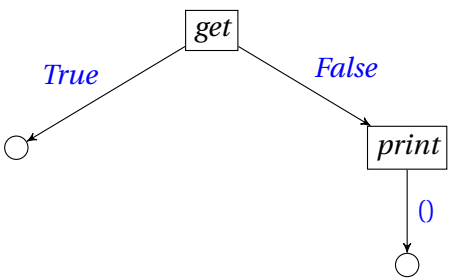
Dependencies

- Computation trees are sequential
- The descendants of a node can depend on the result value of its operation
- We identify three kinds of dependency
  - Control flow: operations depend on prior results
  - Data flow: operation parameters depend on prior results
  - Memory: final return values depend on prior results

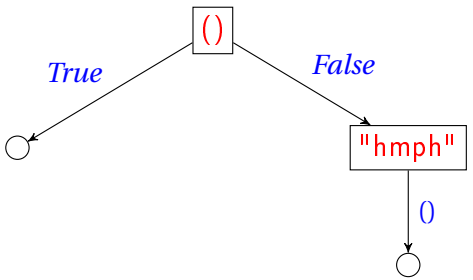
Decomposing example 1 (hmph)



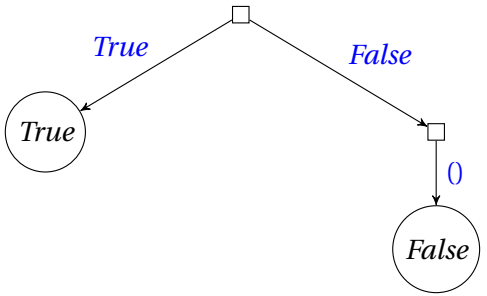
Control flow



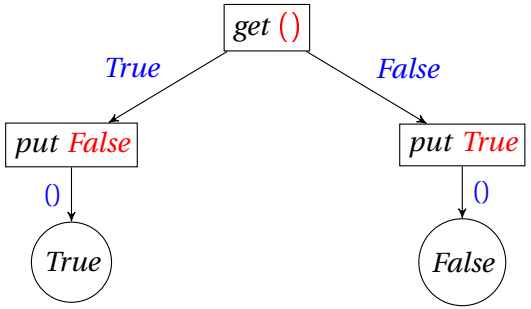
Data flow



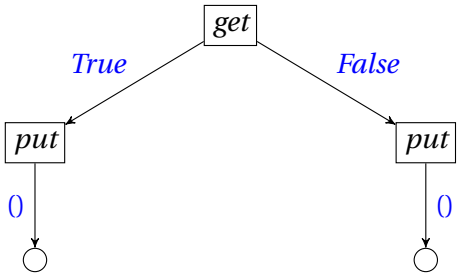
Memory



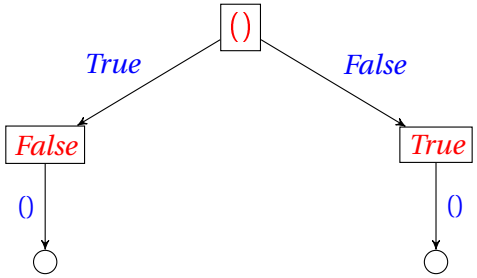
Decomposing example 2 (flip)



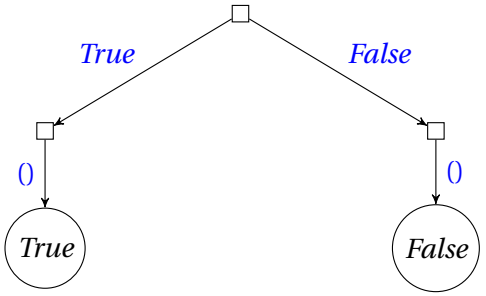
Control flow



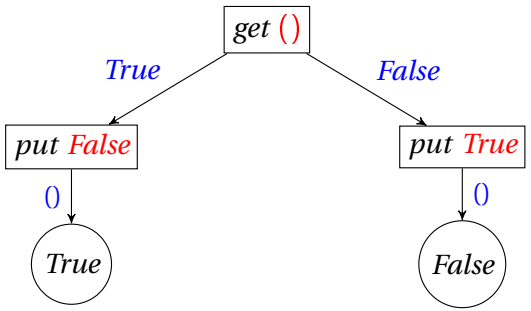
Data flow



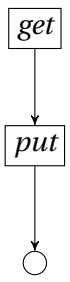
Memory



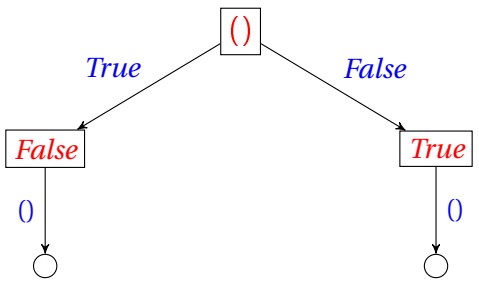
Decomposing example 2 (flip)



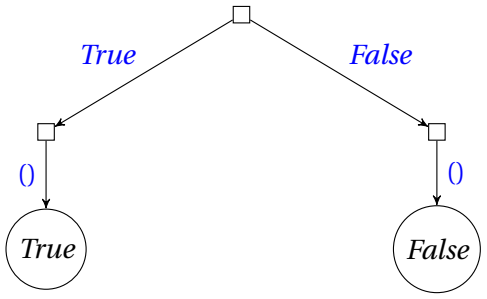
Control flow



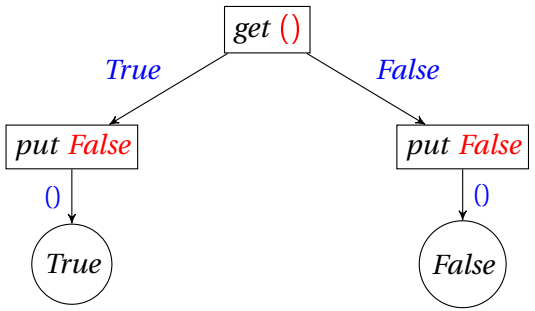
Data flow



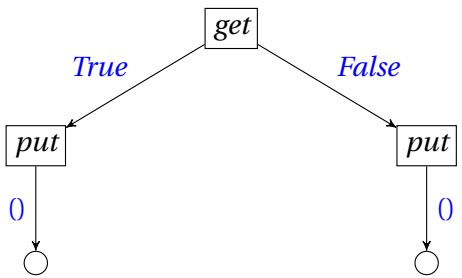
Memory



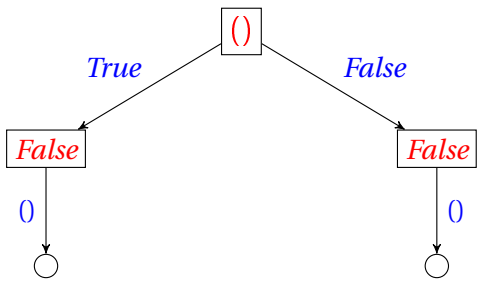
Decomposing example 3 (reset)



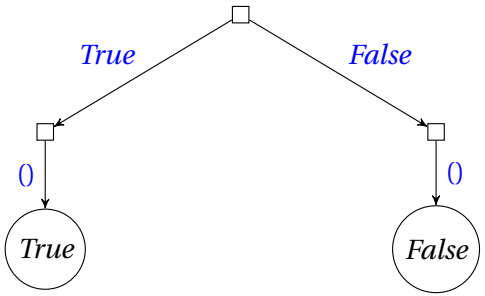
Control flow



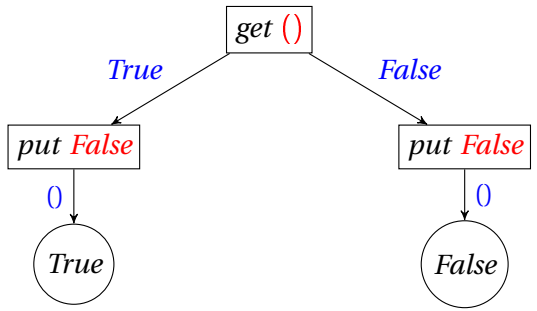
Data flow



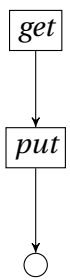
Memory



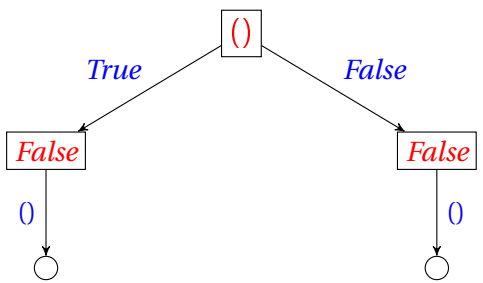
Decomposing example 3 (reset)



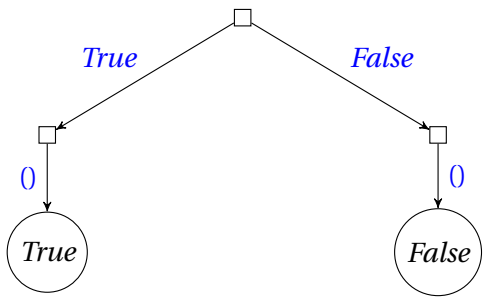
Control flow



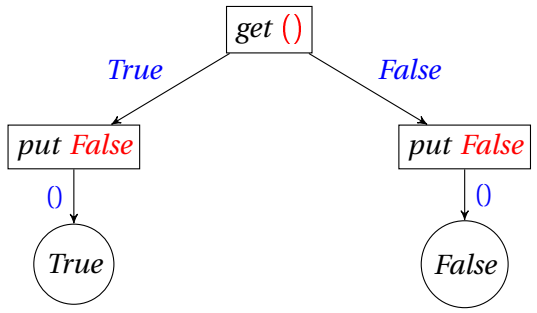
Data flow



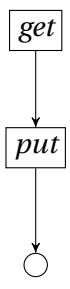
Memory



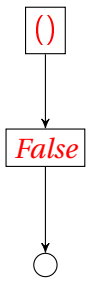
Decomposing example 3 (reset)



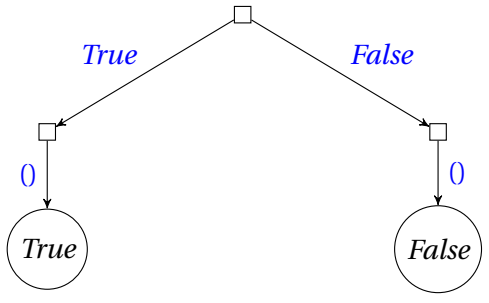
Control flow



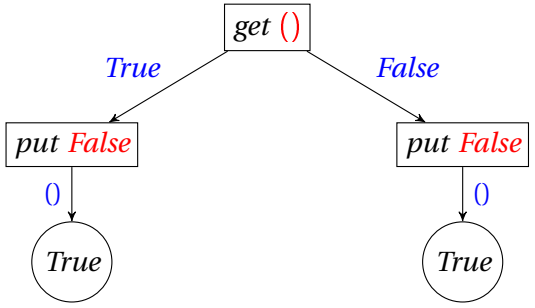
Data flow



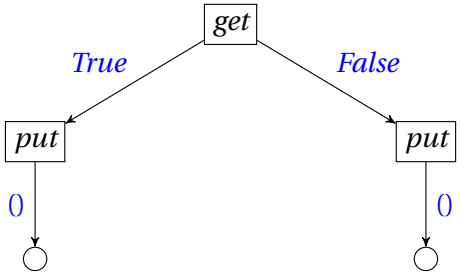
Memory



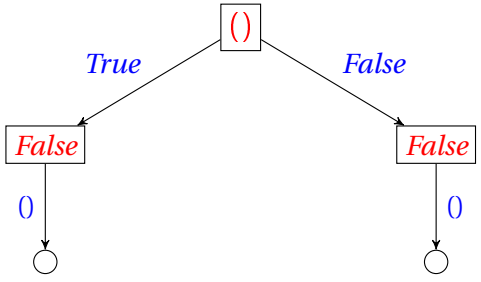
Decomposing example 4 (reset and forget)



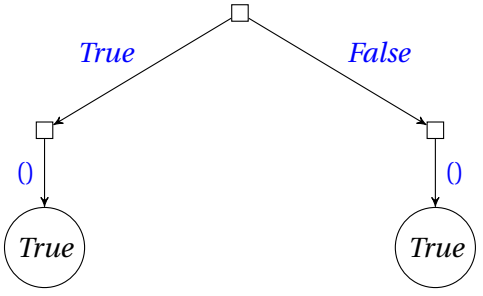
Control flow



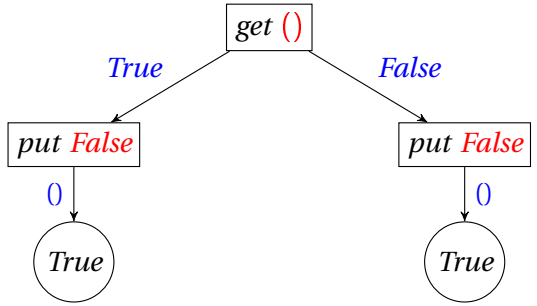
Data flow



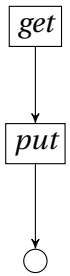
Memory



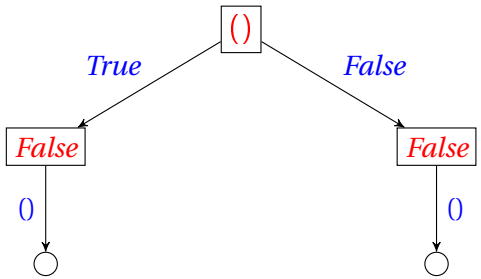
Decomposing example 4 (reset and forget)



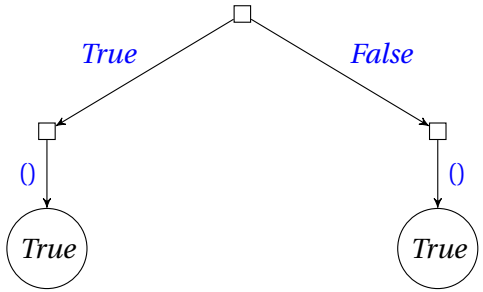
Control flow



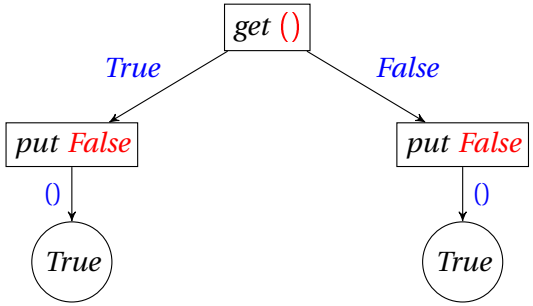
Data flow



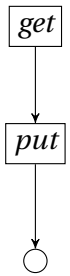
Memory



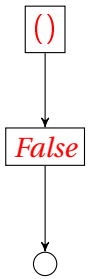
Decomposing example 4 (reset and forget)



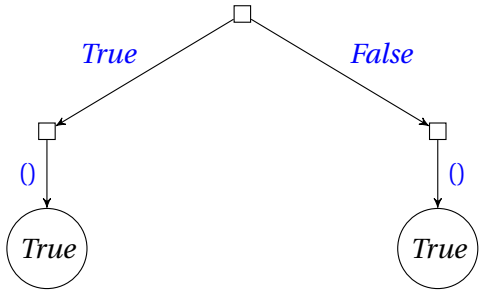
Control flow



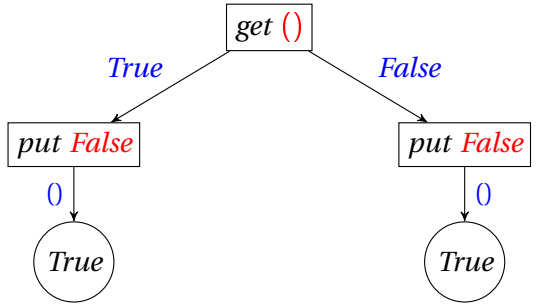
Data flow



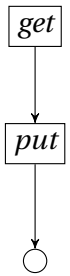
Memory



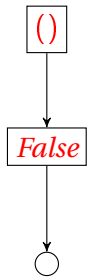
Decomposing example 4 (reset and forget)



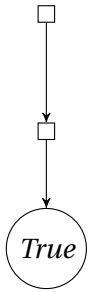
Control flow



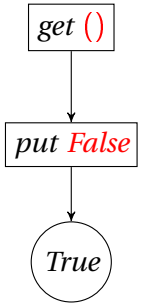
Data flow



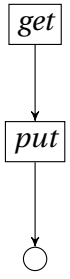
Memory



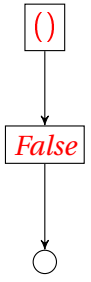
Decomposing example 4 (reset and forget)



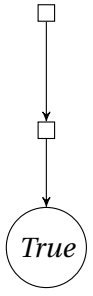
Control flow



Data flow



Memory



Idioms are oblivious, arrows are meticulous, monads are promiscuous

[Lindley, Wadler, and Yallop, MSFP 2008]

computation	control flow	data flow
monad	dynamic	dynamic
arrow	static	dynamic
idiom	static	static

Why are static computations useful?

- additional effect combinations
- hardware implementations
- optimisations

A core effect calculus (λ_{eff})

Types

$$
\begin{array}{lrl}
\text{(values)} & A, B & ::= 1 \mid A_1 \times A_2 \mid 0 \mid A_1 + A_2 \mid \{C\}_E \\
\text{(computations)} & C & ::= [A] \mid A \to C \\
\text{(effect signatures)} & E & ::= \{\mathrm{op} : A \to B\} \uplus E \mid \emptyset \\
\text{(environments)} & \Gamma & ::= x_1 : A_1, \ldots, x_n : A_n
\end{array}
$$

Terms

$$
\begin{array}{lrl}
\text{(values)} & V, W & ::= x \mid () \mid (V_1, V_2) \mid \mathbf{inj}_i\, V \mid \{M\} \\
\text{(computations)} & & \\
& M, N & ::= \mathbf{split}(V, x_1.x_2.M) \mid \mathbf{case_0}(V) \\
& & \mid \ \mathbf{case}(V, x_1.M_1, x_2.M_2) \mid V! \\
& & \mid \ \mathbf{return}\ V \mid \mathbf{let}\ x \leftarrow M\ \mathbf{in}\ N \\
& & \mid \ \lambda x.M \mid M\ V \\
& & \mid \ \mathrm{op}\ V(\lambda x.M)
\end{array}
$$

Value typing rules

$\boxed{\Gamma \vdash V : A}$

$$\frac{(x : A) \in \Gamma}{\Gamma \vdash x : A} \qquad \frac{}{\Gamma \vdash () : 1} \qquad \frac{\Gamma \vdash V_1 : A_1 \qquad \Gamma \vdash V_2 : A_2}{\Gamma \vdash (V_1, V_2) : A_1 \times A_2}$$

$$\frac{\Gamma \vdash V : A_i}{\Gamma \vdash \mathbf{inj}_i\, V : A_1 + A_2} \qquad \frac{\Gamma \vdash_E M : C}{\Gamma \vdash \{M\} : \{C\}_E}$$

Computation typing rules

$\boxed{\Gamma \vdash_E M : C}$

$$\frac{\Gamma \vdash V : A_1 \times A_2 \qquad \Gamma, x_1 : A_1, x_2 : A_2 \vdash_E M : C}{\Gamma \vdash_E \mathbf{split}(V, x_1.x_2.M) : C} \qquad \frac{\Gamma \vdash V : 0}{\Gamma \vdash_E \mathbf{case_0}(V) : C}$$

$$\frac{\Gamma \vdash V : A_1 + A_2 \qquad \Gamma, x_1 : A_1 \vdash_E M_1 : C \qquad \Gamma, x_2 : A_2 \vdash_E M_2 : C}{\Gamma \vdash_E \mathbf{case}(V, x_1.M_1, x_2.M_2) : C}$$

$$\frac{\Gamma \vdash V : A}{\Gamma \vdash_E \mathbf{return}\ V : [A]} \qquad \frac{\Gamma \vdash_E M : [A] \qquad \Gamma, x : A \vdash_E N : C}{\Gamma \vdash_E \mathbf{let}\ x \leftarrow M\ \mathbf{in}\ N : C}$$

$$\frac{\Gamma, x : A \vdash_E M : C}{\Gamma \vdash_E \lambda x.M : A \to C} \qquad \frac{\Gamma \vdash_E M : A \to C \qquad \Gamma \vdash V : A}{\Gamma \vdash_E M\ V : C}$$

$$\frac{\Gamma \vdash V : \{C\}_E}{\Gamma \vdash_E V! : C} \qquad \frac{(\mathrm{op} : A \to B) \in E \qquad \Gamma \vdash V : A \qquad \Gamma, x : B \vdash_E M : C}{\Gamma \vdash_E \mathrm{op}\ V(\lambda x.M) : C}$$

Reductions

$$
\begin{array}{lr}
(\beta.\times) & \mathbf{split}((V_1, V_2), x_1.x_2.M_1) \longrightarrow M[V_1/x_1, V_2/x_2] \\
(\beta.+) & \mathbf{case}(\mathbf{inj}_i\, V, x_1.M_1, x_2.M_2) \longrightarrow M_i[V/x_i] \\
(\beta.\{\}) & \{M\}! \longrightarrow M \\
 & \\
(\beta.[]) & \mathbf{let}\ x \leftarrow \mathbf{return}\ V\ \mathbf{in}\ M \longrightarrow M[V/x] \\
(\beta.\rightarrow) & (\lambda x.M)\ V \longrightarrow M[V/x]
\end{array}
$$

Adding flow effects ($\lambda_{\text{flow-eff}}$)

- Terms and dynamic semantics are unchanged

Types

$$
\begin{array}{lrl}
\text{(values)} & A, B & ::= 1 \mid A_1 \times A_2 \mid 0 \mid A_1 + A_2 \mid \{C\}_E \\
\text{(computations)} & C & ::= [A] \mid A \to C \\
\text{(effect signatures)} & E & ::= \{\mathrm{op} : A \to B\} \uplus E \mid \{\mathrm{f}\} \uplus E \mid \emptyset \\
\text{(flow effects)} & \mathrm{f} & ::= \mathrm{c} \mid \mathrm{d} \mid \mathrm{m} \\
\text{(environments)} & \Gamma & ::= \Gamma, x : A \mid \Gamma, x^\star : A \mid \cdot
\end{array}
$$

- c = control flow, d = data flow, m = memory
- $x^\star : A$ denotes an *active* variable
- $x : A$ denotes an *inactive* variable

Activation

Activating a type environment

$$\begin{aligned} .^\star &= . \\ (\Gamma, x : A)^\star &= \Gamma^\star, x^\star : A \\ (\Gamma, x^\star : A)^\star &= \Gamma^\star, x^\star : A \end{aligned}$$

Only *active* variables can be accessed

$$\textsc{Var}\ \frac{(x^\star : A) \in \Gamma}{\Gamma \vdash x : A}$$

All variables are active in function arguments

$$\textsc{Fun}\ \frac{\Gamma \vdash_E M : A \to C \qquad \Gamma^\star \vdash V : A}{\Gamma \vdash_E M\ V : C}$$

Memory and data flow

Memory

$$\textsc{Return}\ \frac{\Gamma \vdash V : A}{\Gamma \vdash_E \mathbf{return}\ V : [A]} \qquad \textsc{Return}^\star\ \frac{\mathrm{m} \in E \qquad \Gamma^\star \vdash V : A}{\Gamma \vdash_E \mathbf{return}\ V : [A]}$$

Data flow

$$\textsc{Op}\ \frac{(\mathrm{op} : A \to B) \in E \qquad \Gamma \vdash V : A \qquad \Gamma, x : B \vdash_E M : C}{\Gamma \vdash_E \mathrm{op}\ V(\lambda x.M) : C}$$

$$\textsc{Op}^\star\ \frac{\mathrm{d} \in E \qquad (\mathrm{op} : A \to B) \in E \qquad \Gamma^\star \vdash V : A \qquad \Gamma, x : B \vdash_E M : C}{\Gamma \vdash_E \mathrm{op}\ V(\lambda x.M) : C}$$

Control flow (+ data flow + memory)

$$\textsc{Case}\quad \frac{\Gamma \vdash V : A_1 + A_2 \qquad \Gamma, x_1 : A_1 \vdash_E M_1 : C \qquad \Gamma, x_2 : A_2 \vdash_E M_2 : C}{\Gamma \vdash_E \mathbf{case}(V, x_1.M_1, x_2.M_2) : C}$$

$$\textsc{Case}^\star\quad \frac{\mathrm{c}, \mathrm{d}, \mathrm{m} \in E \qquad \Gamma^\star \vdash V : A_1 + A_2 \qquad \Gamma, x_1 : A_1 \vdash_E M_1 : C \qquad \Gamma, x_2 : A_2 \vdash_E M_2 : C}{\Gamma \vdash_E \mathbf{case}(V, x_1.M_1, x_2.M_2) : C}$$

$$\textsc{Force}\quad \frac{\Gamma \vdash V : \{C\}_E}{\Gamma \vdash_E V! : C} \qquad \textsc{Force}^\star\quad \frac{\mathrm{c}, \mathrm{d}, \mathrm{m} \in E \qquad \Gamma^\star \vdash V : \{C\}_E}{\Gamma \vdash_E V! : C}$$

Handlers

A handler is an interpreter for abstract computations.

Handler types

$$R ::= A^{E} \Rightarrow^{E'} C \qquad \text{(essentially } \{\{[A]\}_{E} \to C\}_{E'}\text{)}$$

Handlers

$$H ::= \{\mathbf{return}\ x \mapsto M\} \mid H \uplus \{\mathrm{op}\, p\, k \mapsto N\}$$

Handling

$$M ::= \cdots \mid \mathbf{handle}\ M\ \mathbf{with}\ H$$

Handler typing rules

$$\boxed{\Gamma \vdash H : A^{E} \Rightarrow^{E'} C}$$

$$\frac{\begin{array}{c} E = \{\mathrm{op}_i : A_i \to B_i\}_i \\ H = \{\mathbf{return}\ x \mapsto M\} \uplus \{\mathrm{op}_i\ p\ k \mapsto N_i\}_i \\ [\Gamma, p : A_i, k : \{B_i \to C\}_{E'} \vdash_{E'} N_i : C]_i \qquad \Gamma, x : A \vdash_{E'} M : C \end{array}}{\Gamma \vdash H : A^{E} \Rightarrow^{E'} C}$$

$$\frac{\Gamma \vdash_{E} M : [A] \qquad \Gamma \vdash H : A^{E} \Rightarrow^{E'} C}{\Gamma \vdash_{E'} \mathbf{handle}\ M\ \mathbf{with}\ H : C}$$

Problem: if we generalise to non-monadic computations then the continuation argument k need not inhabit the usual function space.

Handler reductions

hoisting frames

$$\mathscr{H} ::= \textbf{let } x \leftarrow [\,] \textbf{ in } N \mid [\,]\ V$$

$$(\mathit{hoist}.\mathrm{op})\quad \frac{x \notin FV(\mathscr{H})}{\mathscr{H}[\mathrm{op}\ V(\lambda x.M)] \longrightarrow \mathrm{op}\ V(\lambda x.\mathscr{H}[M])}$$

$$(\mathit{handle}.[])\quad \frac{H^{\textbf{return}} = \lambda x.M}{\textbf{handle}\ (\textbf{return}\ V)\ \textbf{with}\ H \longrightarrow M[V/x]}$$

$$(\mathit{handle}.\mathrm{op})\quad \frac{H^{\mathrm{op}} = \lambda p\,k.N \qquad x \notin FV(H)}{\textbf{handle}\ \mathrm{op}\ V(\lambda x.M)\ \textbf{with}\ H \longrightarrow N[V/p, \{\lambda x.\textbf{handle}\ M\ \textbf{with}\ H\}/k]}$$

Handler typing with flow effects (half-baked)

$\boxed{\Gamma \vdash H : A^{E} \Rightarrow^{E'} C}$ ($\Rightarrow$ is a meta variable)

$$\frac{\begin{array}{c} E = \{\mathrm{op}_i : A_i \to B_i\}_i \uplus \{\mathrm{f}_j\}_j \\ H = \{\mathbf{return}\ x \mapsto M\} \uplus \{\mathrm{op}_i\ p_i\ k_i \mapsto N_i\}_i \\ [\Gamma, p_i^\star : A_i, k_i^\star : \{B_i \Rightarrow C\}_{E'} \vdash_{E'} N_i : C]_i \qquad \Gamma, x^\star : A \vdash_{E'} M : C \end{array}}{\Gamma \vdash H : A^{E} \Rightarrow^{E'} C}$$

$$\frac{\Gamma^\dagger \vdash_E M : [A] \qquad \Gamma \vdash H : A^{E} \Rightarrow^{E'} C}{\Gamma \vdash_{E'} \mathbf{handle}\ M\ \mathbf{with}\ H : C}$$

Flushing a type environment

$$\begin{aligned} .^\dagger &= . \\ (\Gamma, x : A)^\dagger &= \Gamma^\dagger \\ (\Gamma, x^\star : A)^\dagger &= \Gamma^\dagger, x^\star : A \end{aligned}$$

Future work

- Embeddings of existing formalisms into $\lambda_{\text{flow-eff}}$
- Pin down handlers for $\lambda_{\text{flow-eff}}$
- Consider other combinations of flow effects (e.g. does dynamic control flow + static data flow make sense?)
- Practical examples with handlers
- Equations

Trees picture: wili_hybrid of Flickr