Lazy interactions – back to the future

Simon Thompson, University of Kent
System.IO.interact :: (String -> String) -> IO ()
System.IO.interact :: (String -> String) -> IO ()

interact f = do s <- getContents
               putStrLn (f s)
\[ f :: (\text{Input} \rightarrow \text{Output}) \]

The output of the program is a function of its input.
\[ f :: (\text{Input} \rightarrow \text{Output}) \]

\[
\begin{align*}
3 & \quad (2, 23) \\
23 & \quad (1, 68) \\
45 & \\
67 & \quad (0, 135)
\end{align*}
\]
\[ f :: (\textbf{Input} \rightarrow \textbf{Output}) \]

3
23
(2,23)
45
(1,68)
67
(0,135)
\[ f :: (\text{Input} \rightarrow \text{Output}) \]

3
23
(2,23)
45
(1,68)
67
(0,135)

Interaction =
input / output interleaving
$f :: (\text{Input} \rightarrow \text{Output})$

Interaction = input / output interleaving

Interleaving determined by lazy evaluation
The essence of laziness

\[ f \perp = "type now" ++ \perp \]

\[ f ("echo" ++ \perp) = \ldots ++ "ohce" ++ \perp \]
The essence of laziness

\[ f \perp = "type now" \perp \]
\[ f ("echo" \perp) = \ldots \perp \]

Lazy interactions are determined by the behaviour of the function on partial data.
Demo
“Seat of the pants?”

necho ys
  = "Prompt: " ++ [head ys] ++ "\n" ++ necho (tail ys)

vs

necho (x:xs)
  = "Prompt: " ++ [x] ++ "\n" ++ necho xs
“Seat of the pants?”

necho ~(x:xs)
    = "Prompt: " ++ [x] ++ "\n" ++ necho xs

vs

necho (x:xs)
    = "Prompt: " ++ [x] ++ "\n" ++ necho xs
“Seat of the pants?”

Let’s build a model of interactions and how to combine them together …
Back to the future?
YEAR OF PROGRAMMING

The 1987 University of Texas Year of Programming was established early in 1986, in response to a proposal by Profs. J. C. Browne and J. Misra, with the following goals:

1) to advance the art and science of programming by bringing leading scientists together for discussions and collaboration;

2) to disseminate among leading practitioners the best of what has been learned about the theory and practice of programming.
YEAR OF PROGRAMMING

The 1987 University of Texas Year of Programming was established early in 1986, in response to a proposal by Profs. J. C. Browne and J. Misra, with the following goals:

1) to advance the art and science of programming by bringing leading scientists together for discussions and collaboration;

2) to disseminate among leading practitioners the best of what has been learned about the theory and practice of programming.

The tutorial, which provided an introduction to lazy functional programming, consisted of lectures interspersed with programming sessions (conducted with pencil and paper) attended by the lecturers and several teaching assistants. Major topics included data types, polymorphism, recursion and induction, lists, domain theory, program synthesis, and several case studies.
YEAR OF PROGRAMMING

The 1987 University of Texas Year of Programming was established early in 1986, in response to a proposal by Profs. J. C. Browne and J. Misra, with the following goals:

1) to advance the art and science of programming by bringing leading scientists together for discussions and collaboration;

2) to disseminate among leading practitioners the best of what has been learned about the theory and practice of programming.

The tutorial, which provided an introduction to lazy functional programming, consisted of lectures interspersed with programming sessions (conducted with pencil and paper) attended by the lecturers and several teaching assistants. Major topics included data types, polymorphism, recursion and induction, lists, domain theory, program synthesis, and several case studies.

This institute elicited particular enthusiasm among a group of UT graduate students, who circulated among themselves, and subsequently presented to the UT Department of Computer Sciences, a petition calling on the department "to make Functional Programming a more visible priority in the department... [through] recruitment of faculty engaged in research in the field [and] more formal contacts with private research and other departments...".
Research Topics in Functional Programming

The tutorial lectures interspersed with discussions and practice sessions to advance the understanding of functional programming, and put into practice the ideas, examples, and case studies.

Early in 1986, in response to a proposal, the lecturers and participants to disseminate the results and practice of functional programming.

The tutorial goals: to create a community of functional programmers, and to contribute to the dissemination of functional programming ideas and techniques.

This institutional benefit, a petition circulated among the faculty and students of Computer Sciences, was a priority in the department. Encouraging a more visible presence in the field (and among the students, who were the targets of this initiative), this effort was deemed essential for the advancement of the discipline and the institution.
Back to the future?

Does it still make sense now?
Back to the future?

Does it still make sense now?

The power of retrospection …
Back to the future?

Does it still make sense now?

The power of retrospection …

… how we can bring it up to date?
Back to the future?

Does it still make sense now?
The power of retrospection …
… how we can bring it up to date?
… and any missed opportunities?
Back to the future?

Does it still make sense now?
The power of retrospection …
… how we can bring it up to date?
… and any missed opportunities?

Transliterating from Miranda to Haskell
Back to the future?

Does it still make sense now?

The power of retrospection …

… how we can bring it up to date?

… and any missed opportunities?

Transliterate from Miranda to Haskell

Building a formal model of interactions, with some proofs …
(Input,a) -> (Input,b,Output)

Functions with IO side effects

Build by composition
(Input,a) -> (Input,b,Output)

Functions with IO side effects
Build by composition

Interactions with states
State changes type between steps …
… can add, remove, and modify what’s there.
(Input,a) -> (Input,b,Output)

Functions with IO side effects
Build by composition

Interactions with states
State changes type between steps …
… can add, remove, and modify what’s there.
type Interact a b
    = (Input,a) -> (Input,b,Output)

type Condition a
    = (Input,a) -> Bool

type Input    = [String]
type Output   = [String]
How do we put these together?

We assume that all diagrams are well-typed
How do we put these together?

We assume that all diagrams are well-typed.
Sequencing … key combinator

\[
\text{sq} :: \text{Interact } a \ b \rightarrow \text{Interact } b \ c \rightarrow \text{Interact } a \ c
\]

\[
\text{sq } \text{inter1 } \text{inter2 } x
\]
\[
= \text{make\_Output } \text{out1 } (\text{inter2 } (\text{rest},\text{st}))
\]
\[
\quad \text{where } (\text{rest},\text{st},\text{out1}) = \text{inter1 } x
\]
Sequencing ... key combinator

\[
\text{sq} :: \text{Interact } a \ b \rightarrow \text{Interact } b \ c \rightarrow \text{Interact } a \ c
\]

\[
\text{sq \ inter1 \ inter2 \ x} = \text{make\_Output \ out1 \ (inter2 \ (rest, st))}
\]
where \ (rest, st, out1) = \text{inter1} \ x

\[
\text{make\_Output} :: \text{Output} \rightarrow (\text{Input}, a, \text{Output}) \rightarrow (\text{Input}, a, \text{Output})
\]

\[
\text{make\_Output} \ \text{piece} \ (\text{input}, st, out) = (\text{input}, st, \text{piece}++\text{out})
\]
## Sequencing ... key combinator

\[ sq :: \text{Interact } a \ b \rightarrow \text{Interact } b \ c \rightarrow \text{Interact } a \ c \]

\[
sq \ \text{inter1 inter2} \ x \\
= \text{make\_Output \ out1} \ (\text{inter2} \ (\text{rest, st})) \\
\quad \text{where} \ \sim(\text{rest, st, out1}) = \text{inter1} \ x
\]

\[ \text{make\_Output} :: \text{Output} \rightarrow (\text{Input, } a, \text{Output}) \rightarrow (\text{Input, } a, \text{Output}) \]

\[
\text{make\_Output} \ \text{piece} \ (\text{input, st, out}) = (\text{input, st, piece}++\text{out})
\]
Sequencing ... key combinator

\[ sq :: \text{Interact } a \ b \to \text{Interact } b \ c \to \text{Interact } a \ c \]

\[
\begin{align*}
  sq \ \text{inter1} \ \text{inter2} \ x & = \text{make\_Output out1} \ (\text{inter2} \ (\text{rest}, \text{st})) \\
  & \quad \text{where } ~(\text{rest}, \text{st}, \text{out1}) = \text{inter1} \ x
\end{align*}
\]

\[
\text{make\_Output} :: \text{Output} \to (\text{Input}, a, \text{Output}) \to (\text{Input}, a, \text{Output})
\]

\[
\begin{align*}
  \text{make\_Output} \ \text{piece} \ ~(\text{input}, \text{st}, \text{out}) & = (\text{input}, \text{st}, \text{piece}++\text{out})
\end{align*}
\]
Alternation and repetition

\[\text{alt} :: \text{Cond } a \to \text{Interact } a \ b \to \text{Interact } a \ b \to \text{Interact } a \ b\]

\[
\text{alt } \text{cond} \ \text{inter1} \ \text{inter2} \ \text{x} \\
| \text{cond } \text{x} \quad \to \quad \text{inter1 } \text{x} \\
| \text{otherwise} \quad \to \quad \text{inter2 } \text{x}
\]

\[\text{while} :: \text{Cond } a \to \text{Interact } a \ a \to \text{Interact } a \ a\]

\[
\text{while } \text{cond} \ \text{inter} \\
\quad = \ \text{whi} \\
\quad \text{where} \\
\quad \text{whi} = \text{alt } \text{cond} \ (\text{inter} \ \text{`}sq` \ \text{whi}) \ \text{null}
\]
“Passing parameters”

```
pass_param :: Interact a b ->
    (b -> Interact () d) ->
    Interact a d

pass_param int f (input,st)
    = (rest,final,out1++out)
where
    ~(inter1,st1,out1) = int (input,st)
    ~(rest,final,out)  = (f st1) (inter1,())
```
“Passing parameters”

pass_param :: Interact a b -> 
    (b -> Interact b d) -> 
    Interact a d

pass_param int f (input,st) 
  = (rest,final,out1++out) 
  where
    ~(inter1,st1,out1) = int (input,st) 
    ~(rest,final,out)  = (f st1) (inter1,st1)
And some primitives …

We assume that all diagrams are well-typed
And some primitives …

We assume that all diagrams are well-typed.
And some primitives …

\[ \text{apply} \]

\[ \text{write...} \]

\[ \text{read...} \]

\[ \text{forget start change wait} \]

run :: Interact a b -> a -> IO ()
And some primitives ...

write :: String -> Interact a a
write outstring (input,st)  
  = (input,st,[outstring])

run :: Interact a b -> a -> IO ()
run inter st  
  = interact (\chs ->
    case inter (split chs,st) of
    (_,_,out) -> join out ++ "\n"

readin :: Interact () String
readin (input,())
  = (tail input, head input,[])
Demo
Copy input

copy :: Interact () ()

copy = while (_ -> True) (readin `sq` writeout id)
Copy input

copy :: Interact () ()

copy = while (_ -> True) (readin `sq` writeout id)

copy :: Interact () ()

copy = readin `sq` writeout id `sq` copy
Copy input

```haskell
copy :: Interact () ()
copy = while (\_ -> True) (readin `sq` writeout id)
```

```haskell
copy :: Interact () ()
copy = readin `sq` writeout id `sq` copy
```
Input \( N \) then sum \( N \) numbers

\[
\text{collector :: Interact () (Int,Int)}
\]

\[
\text{collector}
\]
\[
= \text{getInt `sq`}
\]
\[
\quad \text{add_val_right 0 `sq`}
\]
\[
\text{while } ((\succ(0::\text{Int})).\text{fst}.\text{snd})
\]
\[
\quad (\text{add_val_left () `sq`}
\]
\[
\quad \text{pass_on getInt `sq`}
\]
\[
\quad \text{apply } (\langle p, (m,s) \rangle \rightarrow (m-1, s+p)) `sq`
\]
\[
\quad \text{wait `sq`}
\]
\[
\quad \text{showkeep}
\]
Input N then sum N numbers

collector :: Interact () (Int,Int)

collector
  =  getInt `sq`
    add_val_right 0 `sq`
  while ((>(0::Int)).fst.snd)
    (add_val_left () `sq`
      pass_on getInt `sq`
      apply (\(p,(m,s))->(m-1,s+p)) `sq`
      wait `sq`
      showkeep)
Input N then sum N numbers

```haskell
collector :: Interact () (Int,Int)

collector
  = getInt `sq`
    add_val_right 0 `sq`
  while ((>(0::Int)).fst.snd)
    (add_val_left () `sq`
      pass_on getInt `sq`
      apply (\(p,(m,s))\)->(m-1,s+p)) `sq`
    wait `sq`
    showkeep)
```
Input \(N\) then sum \(N\) numbers

collector :: Interact () (Int,Int)

collector = getInt `sq`
    add_val_right 0 `sq`
while ((>(0::Int)).fst.snd)
    (add_val_left () `sq`
        pass_on getInt `sq`
        apply (\(p,(m,s))\)->(m-1,s+p)) `sq`
        wait `sq`
        showkeep)
Input N then sum N numbers

collector :: Interact () (Int,Int)

collector
  = getInt `sq`
  add_val_right 0 `sq`
  while (>(0::Int)).fst.snd
    (add_val_left () `sq`
      pass_on getInt `sq`
      apply (\(p,(m,s))\rightarrow(m-1,s+p)) `sq`
      wait `sq`
      showkeep)

  counter
    (counter,sum)
    (counter,sum)
    ((), (counter,sum))
Input N then sum N numbers

collector :: Interact () (Int,Int)

collector
  =  getInt `sq`
    add_val_right 0 `sq`
  while ((>(0::Int)).fst.snd)
    (add_val_left () `sq`
      pass_on getInt `sq`
      apply \((p,(m,s))\)->(m-1,s+p)) `sq`
    wait `sq`
    showkeep)
Input N then sum N numbers

collector :: Interact () (Int,Int)

collector
    = getInt `sq`
    add_val_right 0 `sq`
    while ((>(0::Int)).fst.snd)
        (add_val_left () `sq`
            pass_on getInt `sq`
            apply (
                \(p,(m,s))\)->(m-1,s+p)) `sq`
            wait `sq`
        showkeep)
Input $N$ then sum $N$ numbers

collector :: Interact () (Int,Int)

collector
  = getInt `sq`
    add_val_right 0 `sq`
  while ((>(0::Int)).fst.snd)
    (add_val_left () `sq`
      pass_on getInt `sq`
      apply ($\langle p,(m,s)\rangle\rightarrow(m-1,s+p))` `sq`
      wait `sq`
      showkeep)
Input N then sum N numbers

Make the state abstract, with accessors, mutators etc.

collector :: Interact () (Int,Int)

collector
  =  getInt `sq`
     add_val_right 0 `sq`
   while ((>(0::Int)).fst.snd)
     (add_val_left () `sq`
      pass_on getInt `sq`
      apply (\(p,(m,s))\rightarrow(m-1,s+p)) `sq`
      wait `sq`
      showkeep)
Input \( N \) then sum \( N \) numbers

\[
\text{collectNums} :: \text{Interact \ Int \ Int}
\]

\[
\text{collectNums = addNum}
\]

\[
\quad \text{`pass_param`}
\]

\[
\quad (\n \rightarrow \text{start} \ 0 \ `sq`
\]

\[
\quad \text{seqlist} \ (\text{replicate} \ n \ \text{addNum}) \ `sq`
\]

\[
\quad \text{write} \ "\text{finished}"
\]
Input N then sum N numbers

collectNums :: Interact Int Int

collectNums = addNum
  `pass_param`
  (\n  -> start 0 `sq`
    seqlist (replicate n addNum) `sq`
    write "finished")
Looking back
All the ingredients were there ...

Higher-order functions
Lazy evaluation
Pattern matching
Algebraic data types
Miranda had no \texttt{lambda}, or \texttt{let}.

- A variant of “point-free” style: the need to name abstractions.

Equality overloaded … but not \texttt{show}, …
Few established “design patterns”

The model mixes aspects of

- Monad
- Arrow
- Applicative
The linguistic turn …

Can see this as a shallow embedding of an interaction language.

What would happen if we made that deep?
The linguistic turn ...

Can see this as a *shallow embedding* of an interaction language.

What would happen if we made that deep?

```haskell
data Inter =
  While Cond Inter |
  Alt Cond Inter Inter |
  Seq Inter Inter |
  ...

interpret ::
  Inter -> Interact Int Int
The linguistic turn …

Can see this as a shallow embedding of an interaction language.

What would happen if we made that deep?

Questions of reflection, dependent types etc.

data Inter =
    While Cond Inter |
    Alt Cond Inter Inter |
    Seq Inter Inter |
    ...

interpret ::
    Inter -> Interact Int Int
Types

The fundamental scope of values hasn’t changed …

… but their classifications have.

Roles for e.g. GADTs, dependency here, especially with DSLs?
Time to look at Fudgets again?

[Image]

F, Fudget, et al

The Fudget type

Types

data F a b = F (FSP a b)
  instance FudgetIO F
instance StreamProcIO F

type Fudget a b = F a b
	type FSP a b = SP (FEvent a) (FCommand b)
type TEvent = (Path, FResponse)
type TCommand = (Path, FRequest)
type FEvent a = Message TEvent a

type FCommand a = Message TCommand a

data SP a b

data Message a b = Low a | High b

Description

A fudget is a stream processor with high level streams and low level streams. The high level streams are used for communication between fudgets within a program. The low level streams are for communication with the I/O system.

F hi ho is the Fudget type. hi is the type of high level input messages and ho is the type of high level output messages.

[Diagram]

http://www.altocumulus.org/Fudgets/
And what hasn’t happened?

Routine verification … semantics.

Compilers derived from semantics.

The end of the program as text.

Special purpose parallel hardware.
https://github.com/simonjohnthompson/Interaction