Berry and Curien's Intensional Legacy

Denis Dancanet

School of Computer Science
Carnegie Mellon University

http://www.cs.cmu.edu/~ddr

DIMACS Workshop on Computational Complexity
and Programming Languages
25–26 July 1996

Research Program

- Develop intensional models for programming languages
- Use such models for theoretical and practical results:
  - relative intensional expressiveness results
  - program analysis (e.g., abstract interpretation)

Definitions:

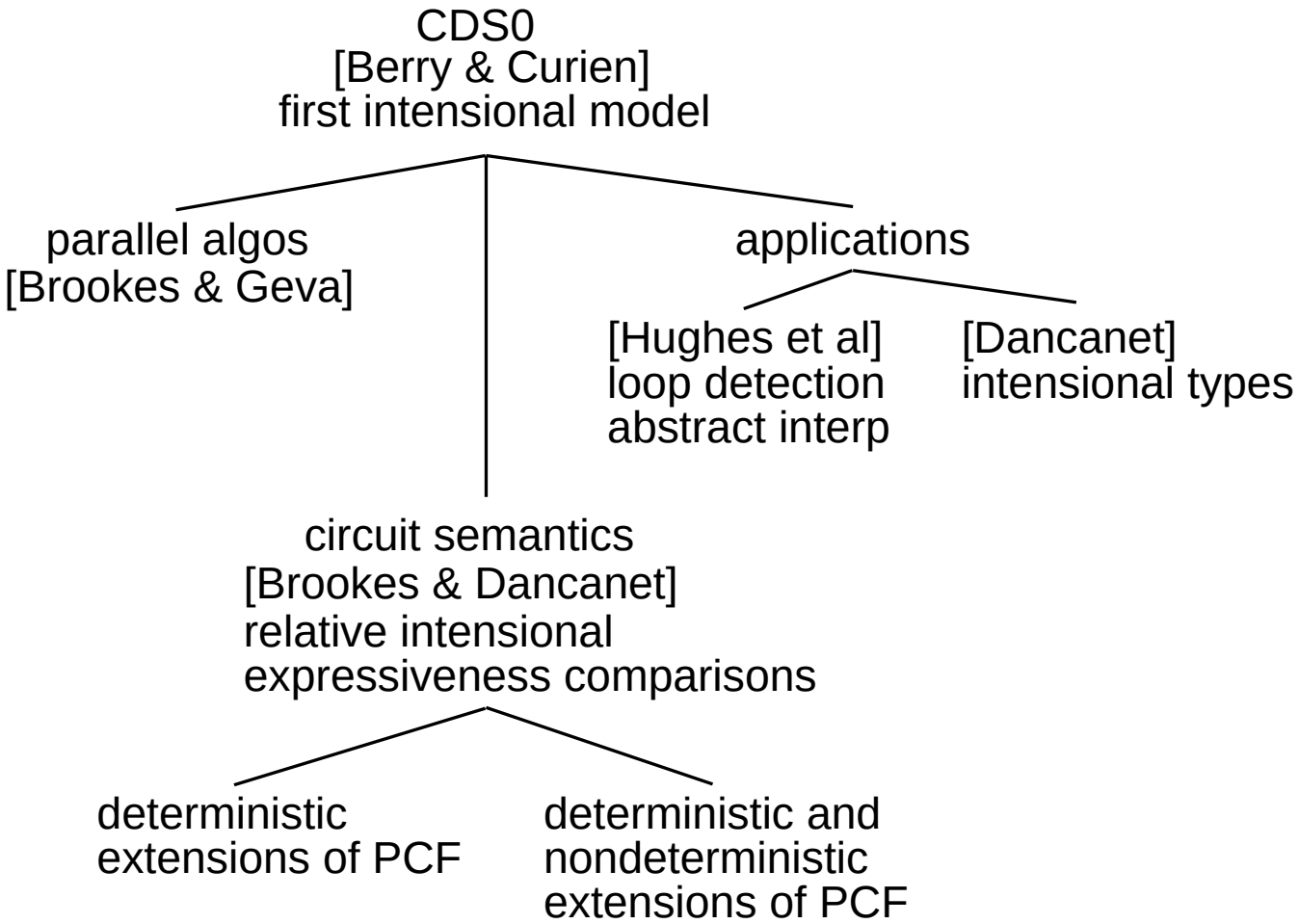
- L_1 is *intensionally* more expressive than L_2 if L_1 can compute all the functions L_2 can, with at least the same asymptotic complexity.
- L_1 is *extensionally* more expressive than L_2 if L_1 can compute all the functions L_2 can.

What is an intensional model?

Meaning of a program could be:

- a function on a richer domain than in extensional semantics, whose structure permits us to deduce information about computation strategy [Colson, Brookes & Geva]
- a pair consisting of a function and an object conveying intensional information; this object could represent the cost of evaluating the function, or could be a function from inputs to costs [Gurr, Rosendahl]
- an algorithm [Berry & Curien]

Outline


CDS0
[Berry & Curien]
first intensional model
parallel algos
[Brookes & Geva]
applications
[Hughes et al]
loop detection
abstract interp
[Dancanet]
intensional types
circuit semantics
[Brookes & Dancanet]
relative intensional
expressiveness comparisons
deterministic
extensions of PCF
deterministic and
nondeterministic
extensions of PCF

PCF

- Simple functional programming language
- Types:

  $\sigma ::= o \mid \iota \mid \sigma \to \sigma$

- Syntax:

  $M ::= c \mid x \mid \lambda x.\ M \mid MM$

- Constants:

  $tt : o$
  $ff : o$
  $n : \iota$ (the integers, $n \geq 0$)
  $isZero? : \iota \to o$
  $+1 : \iota \to \iota$
  $-1 : \iota \to \iota$
  $\supset_o : o \to o \to o \to o$ (boolean conditional)
  $\supset_\iota : o \to \iota \to \iota \to \iota$ (integer conditional)
  $Y_\sigma : (\sigma \to \sigma) \to \sigma$ (one for each σ)
  $= : \iota \to \iota \to o$ (integer equality)

PCF and full abstraction I

- PCF is a paradigmatic sequential programming language
- Operational semantics $\Rightarrow$ observational equivalence
- Denotational semantics $\Rightarrow$ equality of denotations
- Full abstraction:
  observational equivalence $\Leftrightarrow$ equality of denotations
- Standard denotational model of PCF (continuous functions on cpo's) is not fully abstract

PCF and full abstraction II

- $\Omega_\tau = Y_\tau(\lambda f.\ f)$
- ORTEST $= \lambda i.\ \lambda f.$ **if** $(f\ tt\ \Omega)$
  (**if** $(f\ \Omega\ tt)$
  (**if** $(f\ \mathit{ff}\mathit{ff})\ \Omega\ i)$
  Ω
)
  Ω
- ORTEST 0 is operationally equivalent to ORTEST 1
- $\mathcal{D}[\![\text{ORTEST}]\!]\bot =$
  $\lambda v.\ \lambda\Phi.$ if $\Phi = \mathit{por}$ then v else $\bot$
- $\mathcal{D}[\![\text{ORTEST 0}]\!] \neq \mathcal{D}[\![\text{ORTEST 1}]\!]$
- $\mathit{por}\ tt\bot = tt$
  $\mathit{por}\ \bot tt = tt$
  $\mathit{por}\ \mathit{ff}\mathit{ff} = \mathit{ff}$

Concrete data structures

- A cds is like a record:
  - cells
  - values
  - accessibility (enabling) relation
- Example: BOOL

- Event: cell = value
- State: $\{c_1 = v_1, c_2 = v_2, \ldots\}$
- Example:
  States of BOOL = $\{\{\}, \{B = tt\}, \{B = ff\}\}$

Sequential functions

- $F(x)$ = cells filled in state x
- $A(x)$ = cells accessible (enabled and not filled) in state x
- Continuous function $f\colon \mathcal{D}(M) \to \mathcal{D}(M')$ is *sequential* at $x \in \mathcal{D}(M)$ if for any $c' \in A(f(x))$:
  1. either $A(x) = \emptyset$
  2. or $\exists c \in A(x).\ \forall y \in \mathcal{D}(M).$
     $(x \subseteq y\ \&\ c' \in F(f(y))) \Longrightarrow c \in F(y)$
- That $c \in A(x)$ is a *sequentiality index* of f at x for c'
- f is sequential if it is sequential at every $x \in \mathcal{D}(M)$
- Note: *por* is not sequential

Sequential algorithms

- Sequential algorithm:
  1. Abstract view: sequential function + (sequential) computation strategy
  2. Concrete view: a state of a cds of arrow type (*exponentiation* cds)
- Given two cds's, M and M', the exponentiation cds $M \Rightarrow M'$ is:
  1. the cells are of the form xc', where x is a state of M and c' is a cell of M'
  2. the values are of the form $valof\ c$ where c is a cell of M, or $output\ v'$ if v' is a value of M'
- A state of $M \Rightarrow M'$ is a sequential algorithm
- NOT $\in BOOL \Rightarrow BOOL$
- NOT $= \{\{\}B = valof\ B,$
  $\{B = tt\}B = output\ ff,$
  $\{B = ff\}B = output\ tt\}$

CDS0

- Direct implementation of an intensional semantics (sequential algorithms on cds)
- CDS0 was meant to be "assembly" for a higher-level language called CDS
- *Categorical* combinators: application, composition, curry, uncurry, fix, pair, and product
- CDS0 is: lazy, polymorphic, higher-order, functional
- Evaluation model is demand-driven
- Lazy evaluation model allows computing with infinite structures
- Originalities:
  - uniformity of types
  - full abstraction
  - semantics-manipulating programs

CDS0 Examples

```
let bool =
  dcds
    cell B values tt,ff
  end;

 let not = algo
     request B do
       valof B is
         tt : output ff
         ff : output tt
       end
     end
   end;

 let not_state =
   {{}B     = valof B,
   {B=tt}B = output ff,
   {B=ff}B = output tt};
```

Example: And taster

```
let and_taster =
  algo
    request WHICH_AND do
      valof {}B is
        output tt: output IS_NOT_AN_AND
        output ff: output IS_NOT_AN_AND
        valof (B.1):
          valof {(B.1)=tt}B is
            output tt: output IS_NOT_AN_AND
            output ff: output IS_NOT_AN_AND
            valof (B.2):
              valof {(B.1)=tt,(B.2)=tt}B is
                output ff: output IS_NOT_AN_AND
                output tt:
                  valof {(B.1)=tt,(B.2)=ff}B is
                    output tt: output IS_NOT_AN_AND
                    output ff:
                      valof {(B.1)=ff}B is
                        output tt: output IS_NOT_AN_AND
                        output ff: output IS_LEFT_AND
                        valof (B.2):
                          valof {(B.1)=ff,(B.2)=tt}B is
                            output tt: output IS_NOT_AN_AND
                            output ff:
                              valof {(B.1)=ff,(B.2)=ff}B is
                                output tt: output IS_NOT_AN_AND
                                output ff: output IS_LEFT_STRICT_AND
                              end
                          end
                      end
                  end
              end
          end
        valof (B.2): ...
```

Parallel algorithms on cds

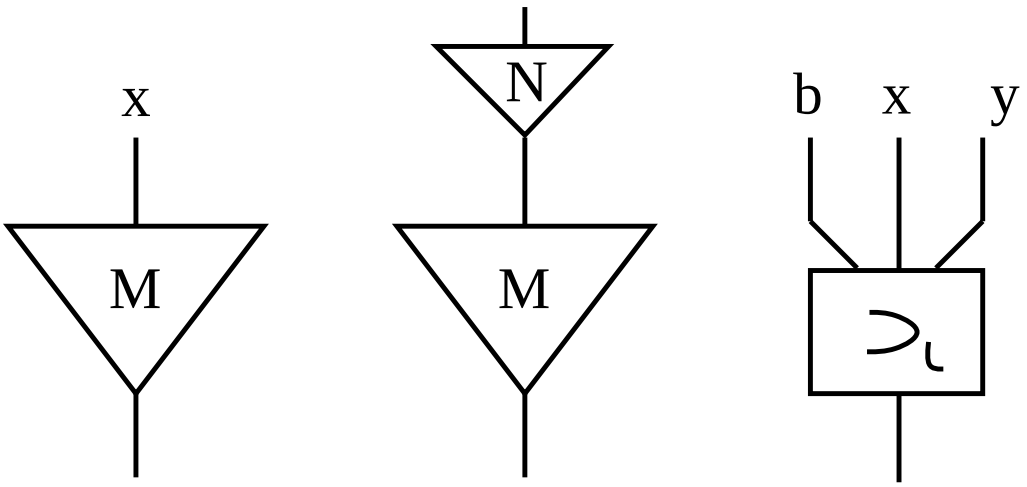
- Generalize *valof* to deterministic parallel *query*
- Example:

```
let por = algo
    request B do
      query {(B.1), (B.2)} is
        {tt, _}  : output tt
        {_, tt}  : output tt
        {ff, ff} : output ff
      end
    end
  end;
```

- Categorical framework for intensional semantics: meaning is function from *computations* to values
- Computation is a comonad with extra structure (co-Kleisli construction)

PCF and circuits 1

- We view PCF programs as circuits:



(a) $\lambda x.\ M$, (b) $(\lambda x.\ M)N$, (c) $\supset_\iota\ b\ x\ y$

- Semantics of circuits based on PCF's operational semantics

PCF and circuits 2

- A circuit is *static* if it represents a non-recursive PCF program
- A circuit is *dynamic* if it represents a recursive PCF program
- The *depth* of a static circuit is equal to the height of the underlying tree
- A circuit is *constant-depth* if it is either static, or a dynamic circuit which does not expand more than a fixed constant number of times (independent of the inputs)
- If two functions can be implemented in terms of each other with constant-depth circuits, we say that the two functions are *intensionally equivalent*

Parallel extensions of PCF

- Parallel-or:
  $por\ :\ o \to o \to o$
  $por\ tt\ \bot = tt$
  $por\ \bot\ tt = tt$
  $por\ ff\ ff = ff$
- Parallel conditionals:
  $pif_\sigma\ :\ o \to \sigma \to \sigma \to \sigma$
  $pif_\sigma\ \bot\ x\ x = x,$
  $pif_\sigma\ tt\ x\ \bot = x,$
  $pif_\sigma\ ff\ \bot\ x = x, \quad \text{for } \sigma = \iota, o$
- It is well known that por, pif_o, pif_ι are extensionally equivalent [Stoughton, Curien]

Intensional equivalence

- **Prop:** por and pif_o are intensionally equivalent
- Proof:

$$
\begin{aligned}
&por = \lambda xy.\ pif_o\ x\ tt\ y, \\
&pif_o = \lambda bxy.\ por_3\ (pand\ b\ x) \\
&\qquad\qquad\qquad\quad (pand\ (not\ b)\ y) \\
&\qquad\qquad\qquad\quad (pand\ x\ y)
\end{aligned}
$$

- Parallel conjunction:

$$pand = \lambda xy.\ not\ (por\ (not\ x)\ (not\ y))$$

Known implementations

- **Prop:** pif_ι can implement pif_o in constant-depth:

  $pif_o = \lambda bxy.\ (= 1\ (pif_\iota\ b\ (\supset_\iota\ x\ 1\ 0)\ (\supset_\iota\ y\ 1\ 0)))$

- Well known implementation of pif_ι using por:

  $pif_\iota = Y F\, 0,$ where
  $F = \lambda f n b x y.\ \supset_\iota\ (por\ (pand\ (= x\ n)\ (= y\ n))$
  $(pand\ b\ (= x\ n))$
  $(pand\ (not\ b)\ (= y\ n)))$
  n
  $(f\ (+1\ n)\ b\ x\ y)$

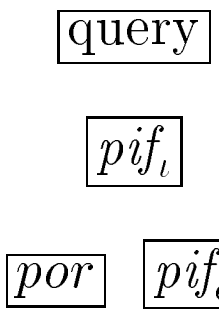
- This is not constant-depth

Intensional separation

- **Lemma 1:** It is not possible to write a program in PCF + *por*, which computes $pif_\iota\ b\ x\ y$ and is of the form $\supset_\iota\ B\ x\ y$, where B is a static circuit yielding a boolean
- **Lemma 2:** It is not possible to write a program in PCF + *por*, which computes $pif_\iota\ b\ x\ y$ and is of the form $\supset_\iota\ B\ N_1\ N_2$, where B, N_1, N_2 are static circuits yielding a boolean and two integers respectively
- **Prop:** PCF + *por* cannot implement pif_ι with a constant-depth circuit

Query is more powerful

- Assume existence of binary addition operation in PCF
- **Prop:** PCF + pif_ι cannot implement n-ary addition in depth $log\ n$
- **Corollary:** PCF + por cannot implement n-ary addition in depth $log\ n$
- Hierarchy of intensional expressiveness for deterministic parallel constructs:



- We have exhibited languages extensionally but not intensionally equivalent

Recursion-free PCF

- A slightly modified PCF
- Types:

  $\sigma ::= o \mid \iota \mid \sigma \to \sigma$

- Syntax:

  $M ::= c \mid x \mid \lambda x.\ M \mid MM$

- Constants:

  $tt, ff : o$
  $n : \iota$
  $=, <, >, \leq, \geq : \iota \to \iota \to o$
  $+, - : \iota \to \iota \to \iota$
  $\supset_\sigma : o \to \sigma \to \sigma \to \sigma$ (conditional, $\sigma \in \{o, \iota\}$)
  $\Omega^\sigma : \sigma$ (undefined elements)

- Standard denotational semantics for PCF:
  $\mathcal{D}$: Terms $\to$ Environments $\to \cup D_\sigma$

Deterministic query

- Syntax:
  query : $(\sigma_1 \times \cdots \times \sigma_n) \to \tau$
  query $(x_1^{\sigma_1}, \ldots, x_n^{\sigma_n})$ is
  $p_1 \Rightarrow M_1^\tau$
  $\ldots$
  $p_k \Rightarrow M_k^\tau$
- Consistent inputs produce same output:
  if $p_i \Uparrow p_j$ then $\mathcal{D}[\![M_i]\!] = \mathcal{D}[\![M_j]\!]$.
- DPCF = PCF + deterministic query

Deterministic query: Semantics

$$\mathcal{D}_{pat} : \text{Patterns} \to \text{Environments} \to (D_{Bool} \times \text{Environments})$$

$$\mathcal{D}[\![Q]\!]\rho = \text{amb}_k(\mathcal{D}[\![\vec{x} \text{ is } p_1 \Rightarrow M_1^\tau]\!]\rho, \ldots, \mathcal{D}[\![\vec{x} \text{ is } p_k \Rightarrow M_k^\tau]\!]\rho)$$

$$\mathcal{D}[\![\vec{x} \text{ is } p \Rightarrow M^\tau]\!]\rho = \begin{cases} \mathcal{D}[\![M^\tau]\!](\rho \mid \mu), & \text{if } \mathcal{D}_{pat}[\![\vec{x} \text{ is } p]\!]\rho = (tt, \mu) \\ \bot, & \text{if } \mathcal{D}_{pat}[\![\vec{x} \text{ is } p]\!]\rho = (ff, \mu) \end{cases}$$

$$\mathcal{D}_{pat}[\![(x_1^{\sigma_1}, \ldots, x_n^{\sigma_n}) \text{ is } (e_1^{\sigma_1}, \ldots, e_n^{\sigma_n})]\!]\rho =$$
$$(b_1 \wedge \cdots \wedge b_n, \mu_1 \mid \cdots \mid \mu_n), \text{where } (b_i, \mu_i) = \mathcal{D}_{pat}[\![x_i^{\sigma_i} \text{ is } e_i^{\sigma_i}]\!]\rho$$

$$\mathcal{D}_{pat}[\![x^\sigma \text{ is } e^\sigma]\!]\rho = \begin{cases} (\mathcal{D}[\![x^\sigma]\!]\rho = \mathcal{D}[\![e^\sigma]\!]\rho, \bot), & \text{if } e^\sigma \text{ is closed} \\ (tt, y^\sigma \mapsto x^\sigma), & \text{if } e^\sigma \equiv y^\sigma \\ (tt, \bot), & \text{if } e^\sigma \equiv _ \end{cases}$$

McCarthy's ambiguity operator:
$amb(\bot, x) = x$,
$amb(x, \bot) = x$,
$amb(x, y) = x$ **or** y.

Nondeterministic query

- Example:
  $not_\perp \equiv \lambda x.$ query (x) is
  $$\begin{array}{l} \quad tt \Rightarrow \mathit{ff} \\ \mid \mathit{ff} \Rightarrow \mathit{ff} \\ \mid _ \Rightarrow tt \end{array}$$
- Under the semantics of deterministic query, nondeterministic query is not intensionally more expressive.

Nondeterministic query: Semantics

- Semantics:

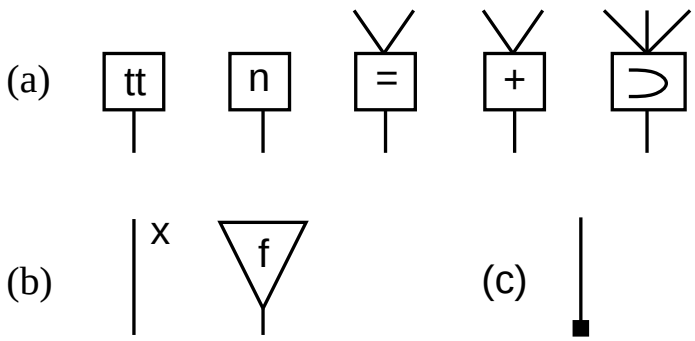
$$\mathcal{D}_{pat}[\![x^\sigma \text{ is } __{ij}]\!]\rho = \begin{cases} (\text{not } (\text{poll } x^\sigma), \bot), \\ \quad \text{if exhausted}(x^\sigma, i, j, p_1, \ldots, p_k, \rho) \\ (tt, \bot), \\ \quad \text{otherwise} \end{cases}$$

$$\text{exhausted}(x^o, i, j, p_1, \ldots, p_k, \rho) = \begin{cases} tt, & \text{if Prop}_o \\ ff, & \text{otherwise} \end{cases}$$

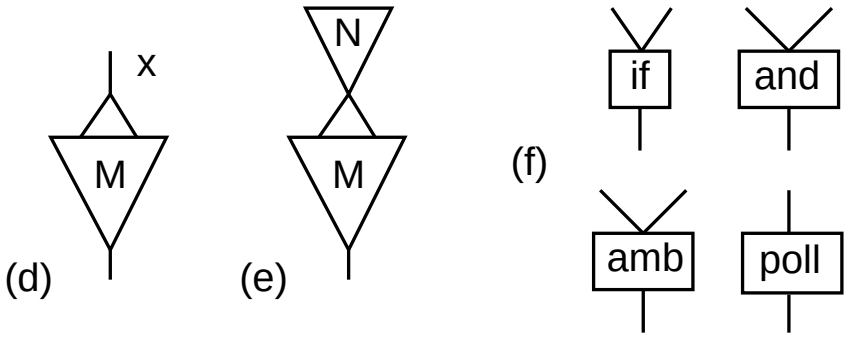
$$\text{exhausted}(x^\iota, i, j, p_1, \ldots, p_k, \rho) = \begin{cases} tt, & \text{if Prop}_\iota \\ ff, & \text{otherwise} \end{cases}$$

- $\text{Prop}_o \equiv$
  $((\exists l, m.\ \mathcal{D}[\![e_{lj}]\!]\rho = tt \wedge \mathcal{D}[\![e_{mj}]\!]\rho = ff) \vee$
  $(\exists l.\ \mathcal{D}[\![e_{lj}]\!]\rho = y^o)) \wedge (p_1 \backslash j \Uparrow \cdots \Uparrow p_k \backslash j)$
- $\text{Prop}_\iota \equiv (\exists l.\ \mathcal{D}[\![e_{lj}]\!]\rho = y^\iota) \wedge (p_1 \backslash j \Uparrow \cdots \Uparrow p_k \backslash j)$
- NPCF = PCF + nondeterministic query

Circuit semantics: Building blocks

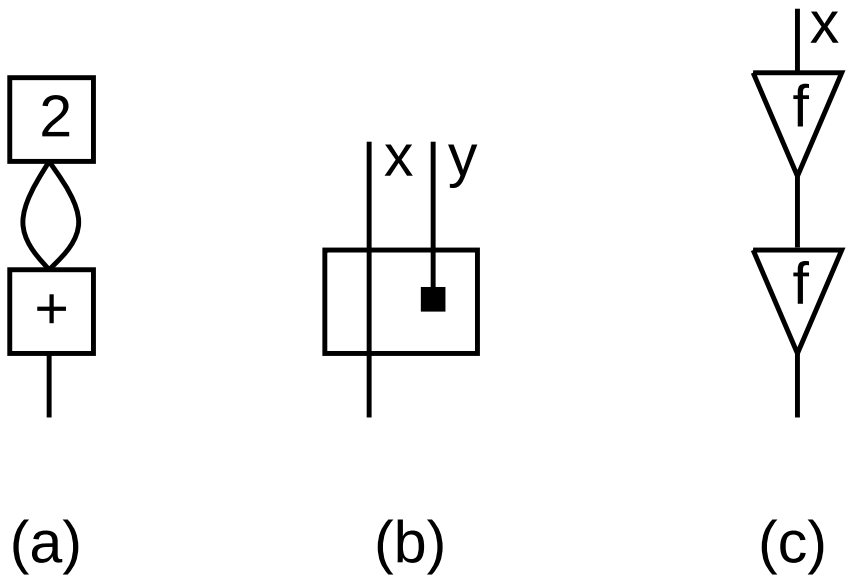


(a) constants, (b) variables, (c) ignored inputs



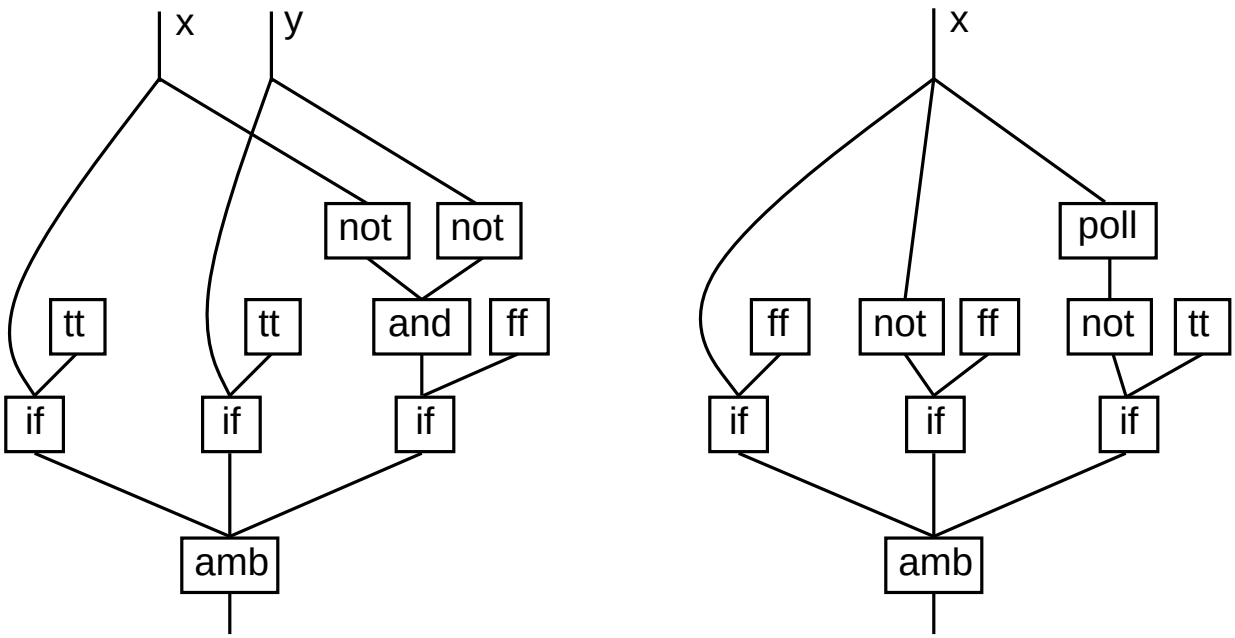
(d) abstraction, (e) application, (f) query

Circuit semantics: PCF examples



(a) $(\lambda x.\ x + x)2$, (b) $\lambda xy.\ x$, (c) $\lambda fx.\ f(fx)$

Circuit semantics: Query examples



(a) por, (b) $not_{\perp}$

Circuit semantics: Properties I

- Define extended semantics for PCF:
  $\mathcal{E}$: Terms $\rightarrow$ Environments $\rightarrow (\cup D_\sigma \times \text{S-Terms}^2)$

  $depth(M_1 + M_2) = 1 + \max(depth(M_1), depth(M_2))$
  $depth(x^\sigma) = x^\sigma$
  $depth(M_1\ M_2) = depth(M_1)\ depth(M_2)$
  $depth(\lambda x^\sigma.\ M) = \Lambda x^\sigma.\ depth(M)$

- For a PCF program M, $\mathcal{E}[\![M]\!]\perp = (v, d, s)$ if and only if the circuit representing M has depth d and size s.

Circuit semantics: Properties II

- [Blelloch & Greiner] Define call-by-speculation:
  $\rho, d \vdash M \longrightarrow_{\text{cbs}} v; d', \hat{d}', w$

$$\frac{\begin{array}{c}\rho, d \vdash M_1 \longrightarrow_{\text{cbs}} v_1; d_1, \hat{d}_1, w_1 \\ \rho, d \vdash M_2 \longrightarrow_{\text{cbs}} v_2; d_2, \hat{d}_2, w_2\end{array}}{\rho, d \vdash M_1 + M_2 \longrightarrow_{\text{cbs}} \begin{array}{l} v_1 + v_2; 1 + \max(d_1, d_2), \\ 1 + \max(\hat{d}_1, \hat{d}_2), 1 + w_1 + w_2\end{array}}$$

- If $\mathcal{E}[\![M]\!]\perp = (v, d, s)$, and
  $\perp, 0 \vdash M \longrightarrow_{\text{cbs}} v; t, \hat{t}, w$, for a program M, then:

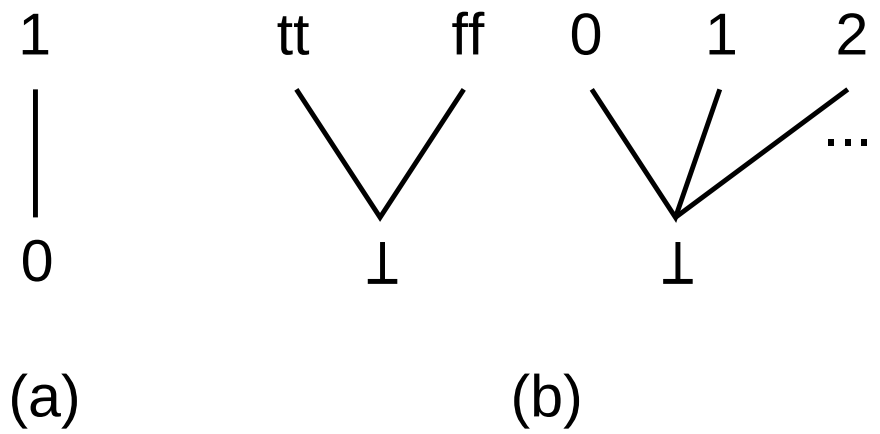
  1. $s \leq w$;
  2. $d \leq \hat{t}$;
  3. if M is conditional-free, then $d = t$.

Method and metric

- Circuit semantics defines parallel complexity: parallel time and parallel work
- Notions of expressiveness:
  - time-expressiveness
  - work-expressiveness
- Assume hardware that can detect undefined inputs
- Question: On such hardware, is NPCF intensionally more time- or work-expressive than DPCF?

Boolean circuits

- Two bases for boolean circuits:
  - monotone: AND, OR
  - De Morgan: AND, OR, NOT
- Monotonicity:



(a) circuit domain, (b) PCF domains

Connection with PCF

- Given a monotone (De Morgan) circuit computing f we can construct from it a DPCF (NPCF) program computing f_{PCF}, whose dimensions in the circuit semantics are the same.
- If DPCF can compute $f : \{\bot, tt\}^n \to tt$, then monotone circuits can compute f_{Bool} with a circuit whose dimensions are the same as the circuit semantics of the DPCF program.

Applications

- [Tardos] There exists a polynomial time computable monotone function whose monotone complexity is exponential.
- There exists a function computable by an NPCF program with polynomial work, but for which the best DPCF program does exponential work.
- [Raz & Wigderson] There is a monotone function in NC^1 that has no monotone NC circuits.
- There exists a function computable by an NPCF program in logarithmic time, and for which the best DPCF program takes more than polylogarithmic time.

Practical applications of CDS0

- [Hughes et al] loop detection and abstract interpretation for higher-order functional language
- We propose refinement type inference
- Example: boolean negation

input	output
{B = tt}	{B = ff}
{B = ff}	{B = tt}

- `not : true -> false /\ false -> true`

Left and

```
let left_and =
  algo
    request B do
      valof (B.1) is
        tt: valof (B.2) is
              tt: output tt
              ff: output ff
            end
        ff: output ff
      end
    end
  end;


left_and : /\[true * true -> true,
              true * false -> false,
              false * bool -> false]
```

And taster

```
and_taster :
  /\[(bool * bool -> true) -> notand,
     (bool * bool -> false) -> notand,
     (true * bool -> true) -> notand,
     (false * bool -> true) -> notand,
     (true * true -> false) -> notand,
     /\[true * true -> true, true * false -> true] -> notand,
     /\[true * true -> true, true * false -> false, false * bool -> true] -> notand
     /\[true * true -> true, true * false -> false, false * bool -> false] -> isand,
     /\[true * true -> true, true * false -> false, false * true -> true] -> notand,
     /\[true * true -> true, true * false -> false, false * true -> false,
        false * false -> true] -> notand,
     /\[true * true -> true, true * false -> false, false * true -> false,
        false * false -> false] -> isand,

     ...

     ]
```

Directions for future research

- More intensional expressiveness results?
- Is the circuit semantics useful to complexity theorists?
- Refinement type inference for a message passing parallel language
- Is intensionality inherently expensive to use?
- What is intensional programming good for?