Circuit Semantics and Intensional Expressivity

Stephen Brookes, Denis Dancanet [1]

School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213

Abstract

We introduce a new denotational semantics for functional programming languages and use it to reason about intensional aspects of programs. Circuit semantics associates a gate with each basic construct of the language, and takes the meaning of a program to be a circuit. The dimensions of the circuit enable reasoning about running time and work required for execution. We use circuit semantics to obtain relative intensional expressiveness results between various deterministic and nondeterministic parallel extensions of PCF.

1 Introduction

We are interested in establishing relative *intensional* expressiveness results for programming languages. Most work in the past has focused on *extensional* expressiveness: Language L_1 is *extensionally more expressive* than L_2 if L_1 can compute all the functions computable in L_2. We say that language L_1 is *intensionally more expressive* than L_2, if L_1 can compute all the functions computable in L_2, with at least the same asymptotic complexity. The notions of complexity we concentrate on are time and work. Note that there has been a lot of work comparing the intensional expressiveness of different *models* of computation. For instance, allowing only a single tape for a Turing machine can square the time necessary to recognize a language versus a two-tape Turing machine [18]; and there are certain problems for which there exist faster CRCW PRAM algorithms than EREW PRAM algorithms [10]. Our goal is to compare programming languages, not their underlying computation models. The distinction can become rather fuzzy however, as the second part of this paper shows, where we are forced to make a hardware assumption in order to be able to compare two constructs.

[1] Corresponding author. Email: `dancanet@cmu.edu`

The paper consists of two parts, both concerned with establishing relative intensional expressiveness results for parallel extensions of the functional language PCF [26], and both utilizing circuit semantics as the main tool. Circuit semantics associates a gate with each basic construct of the language, and takes the meaning of a program to be a circuit. The dimensions of the circuit enable reasoning about running time and work required for execution. In the first part of the paper, we compare four deterministic extensions of PCF: parallel-or, parallel conditionals on booleans and integers, and deterministic query [6]. Query is a simple parallel construct which allows several expressions to be evaluated at the same time. To aid us in this comparison we introduce a naïve version of the circuit semantics [5], which enables us to talk about relative *depth* of an implementation. This notion is good enough to produce a hierarchy of intensional expressiveness: query is the most powerful, followed by parallel conditional on integers, while parallel-or and parallel conditional on booleans are equivalent and the weakest.

In the second part of the paper, we compare deterministic query with a nondeterministic version [12]. We refine the circuit semantics to allow us to talk about parallel time and parallel work required for execution, and we establish connections between the size and depth of a circuit representing a parallel PCF program to the time and work required to execute it under call-by-speculation [19], parallel call-by-value, and parallel eager evaluation. We also relate the circuit dimensions of a program to the time and number of processors required to execute it in the PRAM model.

In order to be able to compare the two versions of query, we are forced to make a hardware assumption which is equivalent to having the ability to detect undefined inputs. This makes a subset of the programs using nondeterministic query return a deterministic result. The assumption is reasonable from a practical point of view and has been used in various studies of consensus problems in distributed systems [14]. The effect of this assumption is to render our problem similar to one from computational complexity, that of comparing monotone and De Morgan boolean circuits. It turns out that parallel PCF programs are intensionally equivalent to boolean circuits for a certain class of functions involving undefined inputs. This connection allows us to use strong results from complexity theory to establish intensional expressiveness results.

1.1 Related work

Several authors [29,28,30,35] have used *profiling semantics*, *i.e.,* operational semantics augmented with time and work information, to perform automatic complexity analysis. Roe [28] considered a parallel *lenient* language, and Zimmerman [35] a data-parallel language. Hudak and Anderson [19] developed

pomsets as a semantics for parallel functional programs. They were able to distinguish between various evaluation strategies (call-by-value, call-by-name, call-by-need, call-by-speculation). More recently, Blelloch and Greiner [3,16] provided profiling semantics for parallel call-by-value and call-by-speculation. Their aim in [3] was to show that good upper bounds for merging and sorting can be obtained with an implicitly parallel language. The second model [16] was used to prove the efficiency of a particular implementation of call-by-speculation. Both models were related to more traditional parallel models such as the PRAM.

The circuit model we develop in this paper is most closely related to call-by-speculation. The differences are due to the presence of conditionals in the language. In contrast to earlier work, we are interested in proving lower bounds and performing intensional comparisons between parallel languages.

There has been little work on comparing determinism and nondeterminism. Felleisen, in his theory of expressiveness [13], defined a construct c as more expressive than another c' if the translation of a program using c to one using c' requires a *global reorganization* of the program. He showed that adding side-effects to a sequential functional language increases expressive power.

The literature on Id [32], an *implicitly* parallel language, has produced practical examples of comparisons of Id's purely functional core, the extension with I-structures (single-assignment arrays), and the extension with M-structures (arrays with element-level synchronization).

There has been notable work on *extensional* comparisons of merging primitives in dataflow networks [25]. One of the functions considered there is *poll* which checks, without blocking, if an input is present. We make use of *poll* in this paper. However, we are not aware of any relevant *intensional* comparisons of parallel constructs. The general opinion expressed in [24] (and echoed in [17]) seems to be that the main advantage of nondeterminism is in *specifying* a process.

There has been some recent research at the juncture of complexity theory and programming languages theory. For instance, work on machine independent characterizations of complexity classes (Clote's work on NC [7], the characterization of P in terms of bounded linear logic [15] and λ-calculus [22]), Jones's reconstruction of computability and complexity theory from a programming languages perspective [20,21], and Colson's work on the expressiveness of primitive recursion [8,9]. Our work can be seen in the same spirit, as an attempt to bridge the gap between these two areas of theoretical computer science.

1.2 Outline of the paper

Section 2 describes PCF and the deterministic parallel extensions we are comparing. A first version of the circuit semantics is introduced in Section 3 and is used to obtain the first intensional separation results in Section 4. Section 5 describes the recursion-free version of PCF we use for the nondeterministic extension, and introduces the nondeterministic query. The circuit semantics is refined in Section 6 and the method for making the comparison between determinism and nondeterminism is outlined in Section 7. Section 8 shows a connection between our question and circuit complexity, which is used in Section 9 to obtain the separation of deterministic and nondeterministic query. Finally, Section 10 concludes.

2 PCF and deterministic parallel extensions

2.1 PCF

PCF [26] is a paradigmatic sequential functional programming language that has been used in many semantic studies of sequentiality. It is a typed λ-calculus with two ground types, booleans (o) and integers (ι) and the following set of types:

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

The syntax for raw terms is given by the grammar:

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

The constants traditionally included in the language, along with the relevant rules of the operational semantics are shown in Figure 1. For simplicity, we blur the distinction between numerals and integers, and use n to denote both. We omit the type inference rules, which are standard.

In addition to the standard constants listed in Figure 1, we assume the existence of a constant-time equality test for integers:

$$= : \ \iota \to \iota \to o$$

with the obvious operational semantics. Traditionally, the equality test is implemented using recursion (*cf.* [31]), but this would render some of the issues of interest to us moot.

Let $FV(M)$ stand for the set of free variables of term M. If $FV(M) = \emptyset$ then the term M is *closed*, else it is *open*. The closed terms of ground type are referred to as *programs*.

$tt, ff \ : \ o$		$isZero?\ 0 \to tt$
$n \ : \ \iota$	(integers, $n \geq 0$)	$isZero?\ (n+1) \to ff, \quad \text{for } n \geq 0$
$isZero? \ : \ \iota \to o$		$+1\ n \to n+1, \quad \text{for } n \geq 0$
$+1, -1 \ : \ \iota \to \iota$		$-1\ (n+1) \to n, \quad \text{for } n \geq 0$
$\supset_o \ : \ o \to o \to o \to o$	(bool conditional)	$\supset_\sigma\ tt\ M_\sigma N_\sigma \to M_\sigma, \quad \text{for } \sigma = \iota, o$
$\supset_\iota \ : \ o \to \iota \to \iota \to \iota$	(int conditional)	$\supset_\sigma\ ff\ M_\sigma N_\sigma \to N_\sigma, \quad \text{for } \sigma = \iota, o$
$Y_\sigma \ : \ (\sigma \to \sigma) \to \sigma$	(one for each σ)	$Y_\sigma M \to M(Y_\sigma M)$

Fig. 1. PCF constants and operational semantics

The standard denotational semantics for PCF is given by the semantic function $\mathcal{D}$: Terms $\to$ Environments $\to \bigcup D_\sigma$, where D_σ is a family of domains including the flat domains of booleans (D_{bool}) and integers (D_{int}), and such that $D_{\tau_1 \to \tau_2} = [D_{\tau_1} \to D_{\tau_2}]$, the continuous function space (see [26] for details).

2.2 Parallel-or and parallel conditionals

The parallel extensions of PCF studied in Plotkin's seminal paper [26] are: *por*, pif_o (parallel conditional on booleans), and pif_ι (parallel conditional on integers). The extension of PCF with either of these functions is fully abstract with respect to the standard denotational semantics. The functions are defined as follows:

$por \ : \ o \to o \to o$	$pif_\sigma \ : \ o \to \sigma \to \sigma \to \sigma$
$por\ \bot\ tt = tt$	$pif_\sigma\ \bot\ x\ x = x$
$por\ tt\ \bot = tt$	$pif_\sigma\ tt\ x\ \bot = x$
$por\ ff\ ff = ff$	$pif_\sigma\ ff\ \bot\ x = x$

for $\sigma = \iota, o$. They are known to be extensionally equivalent [11,31], *i.e.,* one can be implemented in terms of another. The question we address is whether they are intensionally equivalent. Interestingly, the answer turns out to be negative.

$$
\begin{array}{ll}
por \equiv \lambda xy.\ \text{query}\ (x,y)\ \text{is} & \text{query}\ :\ (\sigma_1 \times \cdots \times \sigma_n) \to \tau \\
\quad (tt,\ _) \Rightarrow tt & \text{query}\ (x_1^{\sigma_1}, \ldots, x_n^{\sigma_n})\ \text{is} \\
\quad \mid (_,\ tt) \Rightarrow tt & \quad p_1 \Rightarrow M_1^\tau \\
\quad \mid (ff,\ ff) \Rightarrow ff & \quad \ldots \\
 & \quad p_k \Rightarrow M_k^\tau
\end{array}
$$

Fig. 2. (a) parallel-or, (b) query syntax

2.3 Query

Another parallel deterministic extension we are interested in exploring is *query*. Since this is not a well-known construct, we shall devote more space to it. The *query* construct was introduced by Brookes and Geva [6], generalizing the *valof* construct of Berry-Curien sequential algorithms [2] to the parallel setting. Intuitively, a query starts separate processes to evaluate each of its inputs and resumes the main computation when some condition on the values of a finite subset of the sub-computations is met. The addition of query increases the extensional expressiveness of a sequential language, enabling it to express parallel functions like parallel-or (Figure 2(a)). The construct is quite general and can be studied outside of the context of concrete data structures.

The general form of the query syntax is shown in Figure 2(b). The $x_i^{\sigma_i}$ ($\sigma_i \in \{o, \iota\}$) are variables, the p_i are patterns, and the M_i^τ are PCF terms. A pattern is a vector of length n (the number of variables in the query), with each element being either a variable y, a closed term e of ground type, or the "don't care" symbol "_". All of the variables in one pattern must be distinct.

We distinguish between two versions of the query construct: deterministic and nondeterministic. In the first case, we shall require the same output for all *consistent* inputs. Let $p_1 = (v_1, \ldots, v_n)$ and $p_2 = (w_1, \ldots, w_n)$ be two patterns. We call the two patterns *consistent* (written $p_1 \Uparrow p_2$) if $\forall i$, either $\mathcal{D}[\![v_i]\!] \sqsubseteq \mathcal{D}[\![w_i]\!]$ or $\mathcal{D}[\![v_i]\!] \sqsupseteq \mathcal{D}[\![w_i]\!]$. Since the elements of patterns come from flat domains, this formulation of consistency coincides with the conventional notion of "having an upper bound." We extend the standard semantics with $\mathcal{D}[\![_]\!]\rho = \bot$, and note that two variables are always consistent, since they will be bound to the same input value.

A deterministic query has the property that it produces the same output for all consistent inputs, *i.e.,* given two patterns p_i, p_j, if $p_i \Uparrow p_j$ then $\mathcal{D}[\![M_i]\!] = \mathcal{D}[\![M_j]\!]$. The determinism restriction makes it fairly easy to define a semantics,

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

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

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

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

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

Fig. 3. Denotational semantics for deterministic query

shown in Figure 3. We use Q to refer to the general form of query, from Figure 2(b). *Amb* is McCarthy's ambiguity operator [24]:

$$amb(\bot, x) = x, \quad amb(x, \bot) = x, \quad amb(x, y) = x \textbf{ or } y,$$

which behaves essentially like a parallel-or when only one argument is defined, and performs an arbitrary choice between the arguments if both are defined. Amb_k is k-ary *amb*. Because of the determinism constraints, it will always be the case that *amb* will behave deterministically, *i.e.,* if we have $amb(x, y)$ with both x, y defined, then $x = y$. Consequently, the meaning of a query will be a continuous function.

We use the notation $\vec{x}$ for $(x_1^{\sigma_1}, \ldots, x_n^{\sigma_n})$, and $\mid$ for concatenating environments, with the simple properties: $\rho \mid \bot = \bot \mid \rho = \rho$. It is not possible to have multiple bindings for the same variable, because of our requirement that all variables in one pattern should be distinct. $\wedge$ is parallel-and, with the properties: $tt \ \wedge \ tt = tt$, $ff \ \wedge \ \bot = ff$, $\bot \ \wedge \ ff = ff$.

The auxiliary semantic function $\mathcal{D}_{pat}$ defines the meaning of a pattern match. It keeps track of whether the pattern match succeeds and of any bindings generated in the process.

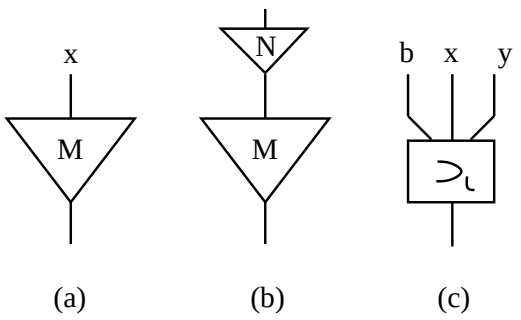

Fig. 4. (a) $\lambda x.\ M$, (b) $(\lambda x.\ M)N$, (c) $\supset_\iota\ b\ x\ y$

3 Circuit semantics: first approach

In order to compare *por*, pif_o, and pif_ι, we find it useful to view PCF programs as circuits. There are several reasons for this. First, it enables us to reason based on the last gate used in the circuit. Viewing a program as a circuit reduces the number of cases we need to consider. Second, the running time of the program *loosely* corresponds to the depth of the circuit. At this stage, we are only interested in the depth of programs, *i.e.,* closed terms of ground type, so we need not worry about complications caused by higher-order terms. Also, a *loose* correspondence is fine, since we only need to distinguish programs that use recursion from programs which do not. And third, circuits provide a visual and intuitive semantics. This is more than a cosmetic point: viewing programs as circuits enables us to find the connection with boolean circuits in the second part of this paper.

The translation from PCF to circuits is simple. Figure 4 shows circuits for function definition, application, and a constant. A function denotes a circuit some of whose inputs are labelled with variables. Application substitutes a value for a variable, or, if we have a whole circuit, connects its output to the respective variable-labelled input. Note that higher-order functions can be treated in this framework as well, by using gates labelled with the function variable inside the circuit (for an example, see Figure 10 in Section 6). There are gates for the various constants. The only interesting case is the Y combinator. It gives rise to a special kind of circuit, a *dynamic circuit*, which can have subparts expanded dynamically as required during computation.

The semantics of circuits is based on PCF's operational semantics. Execution is demand-driven and begins at the output. The last gate in the circuit is activated. This gate may start evaluating one (or more, if it is parallel) of its inputs, leading to activity at further gates, and so on. If the computation terminates, the result will filter down to the output of the last gate.

Definition 1 *A circuit is static if it is the translation of a non-recursive PCF program.*

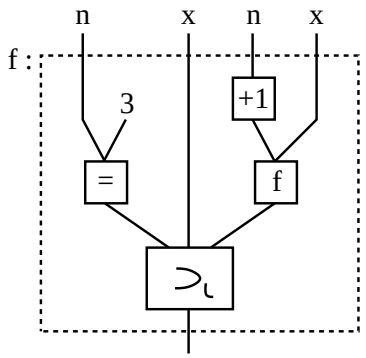

Fig. 5. YF

Definition 2 *A circuit is dynamic if it is the translation of a recursive PCF program.*

A circuit could have several inputs, but it always has just one output, so it is shaped as a tree.

Definition 3 *The depth of a static circuit is equal to the height of the underlying tree.*

Definition 4 *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).*

Example 5 To give an example of dynamic circuits, and to illustrate the difference between constant-depth and non-constant-depth dynamic circuits, consider the following PCF term:

$$F = \lambda fnx.\ \supset_\iota\ (=\ n\ 3)\ x\ (f\ (+1\ n)\ x).$$

Figure 5 shows the circuit denoted by the recursive PCF term YF. We enclose a dynamic circuit in a box with dotted lines, to represent the fact that it can be expanded. The box is labelled with the name of the recursive part. The result of expanding the circuit once is shown in Figure 6.

The program $YF\,n$ for $0 \leq n \leq 3$ gives rise to a constant-depth dynamic circuit, while for $n > 3$ it results in a non-constant-depth dynamic circuit. □

In the following, we are particularly interested in the constant-depth circuits. If two functions can be implemented in terms of each other with constant-depth circuits, we say that the two functions are *intensionally equivalent*.

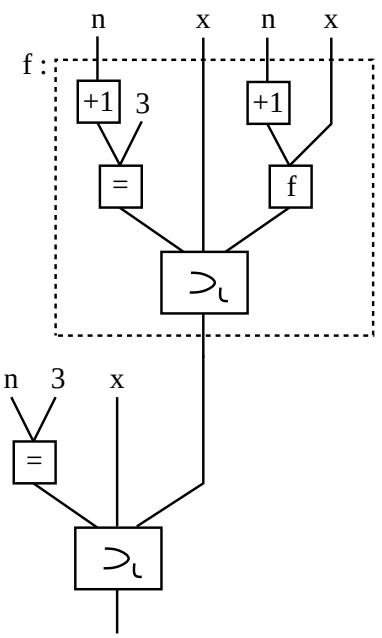

Fig. 6. YF expanded once

4 Intensional separation for deterministic extensions

4.1 pif_ι versus por and pif_o

We begin by reviewing known implementations of the various functions.

Proposition 6 *por and pif_o are intensionally equivalent.*

Proof. We need constant-depth implementations of one in terms of the other. This can be done as follows (*cf.* [31]):

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

where *pand* is the parallel conjunction defined by:

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

and we have generalized *por* to three arguments in the obvious way. □

It is known that pif_ι can implement pif_o (*cf.* [31]):

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

This implementation is also efficient. In view of the previous proposition, it follows that pif_ι can also implement *por* efficiently. However, the converse is false. The problem is that *por* can only start parallel subcomputations on booleans, whereas pif_ι operates in parallel on integers. The standard way of encoding pif_ι with *por* uses recursion (*cf.* [31]):

$$
\begin{array}{l}
pif_\iota = YF\,0, \text{ where} \\
F = \lambda fnbxy.\ \supset_\iota\ (por\ (pand\ (=\ x\ n)\ (=\ y\ n)) \\
\qquad\qquad\qquad\qquad\quad (pand\ b\ (=\ x\ n)) \\
\qquad\qquad\qquad\qquad\quad (pand\ (not\ b)\ (=\ y\ n))) \\
\qquad\qquad\quad n \\
\qquad\qquad\quad (f\ (+1\ n)\ b\ x\ y).
\end{array}
$$

This is clearly inefficient, because of the way the recursion unwinds, checking if x and y are equal to 0 first, then 1, and so on. But we cannot do any better. To show that, we prove first two lemmas which restrict the shape of any program computing pif_ι.

The point of the first lemma is that it is impossible to design boolean circuitry B which chooses between x and y and obeys all the requirements of pif_ι.

Lemma 7 *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.*

Proof. Without loss of generality, the issue is whether it is possible to write a PCF + *por* function B with the following properties:

(i) If b is tt then B is tt,
(ii) If b is ff then B is ff,
(iii) If $(=\ x\ y)$ is tt then B is tt.

Figure 7 shows some of the inputs and corresponding outputs for function B. For simplicity, we assume only b and $(=\ x\ y)$ are used in evaluating B. The same argument can be carried through with additional inputs, since b and $(=\ x\ y)$ *must* be used in evaluating B.

The last line of Figure 7 implies by monotonicity that $B\ ff\ tt = tt$. But this violates the monotonicity condition raised by the second line. Therefore, no program of this form computes $pif_\iota\ b\ x\ y$. □

Our second lemma generalizes the first one.

b	$(= x\ y)$	B
tt	$\perp$	tt
ff	$\perp$	ff
$\perp$	tt	tt

Fig. 7. Requirements for function B

Lemma 8 *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.*

Proof. Intuitively, there are two possibilities for B: either it "chooses" between N_1 and N_2, or it is "hardwired" to always pick one of them. More precisely, we have two cases for the function computed by B:

(i) B is non-constant. Since the program computes $pif_\iota\ b\ x\ y$, the result must be either x or y. There are an infinite number of possible inputs and outputs and N_1, N_2 are static circuits, so it is not possible to hard-code the output. B will sometimes return tt and sometimes ff. There are then three choices for what N_1, N_2 evaluate to:
   (a) They evaluate to x, y, respectively. But this is impossible by Lemma 7.
   (b) They both evaluate to $pif_\iota\ b\ x\ y$. The $\supset_\iota$ gate then does no work. Since N_1, N_2 both compute something of type integer, there are essentially two cases for the last gate used in their construction: (i) $\supset_\iota$ or (ii) $+1$ (-1 is handled similarly). In case (i) apply the same reasoning of this lemma. There cannot be an infinite sequence of $\supset_\iota$ gates which do nothing, since the circuit is static. It is not possible for all $\supset_\iota$ gates to do nothing since the output would then have to be constructed out of $+1$, -1, and the integers, so it would either be hard-coded (and it must work for an infinite number of values), or produce a fixed offset from x or y. The latter case is analogous to case (1a) above, except that the branches evaluate here to a fixed offset of x or y; the same reasoning applies. In case (ii) there cannot only be $+1$ (or -1) gates for the reason outlined above. Also, there can only be a constant number of $+1$ (or -1) in a row before some $\supset_\iota$ is reached, whereupon we can apply the lemma again. By the same reasoning we must at some point encounter case (1a) of the proof.
   (c) One evaluates to $pif_\iota\ b\ x\ y$ and the other to x or y. We apply the same reasoning as in case (1b) to the last gate in the branch evaluating to $pif_\iota\ b\ x\ y$, eventually reaching case (1a).

(ii) B is constant. That means that either N_1 or N_2 must compute $pif_\iota\ b\ x\ y$. Again we have a $\supset_\iota$ gate which does no work. Without loss of generality, assume B is tt, so N_1 always gets chosen. We apply the same reasoning

as in case (1b) to the last gate in N_1 eventually reducing the problem to case (1a).

So our circuit cannot be filled with gates which "do no work." At some point there must be a $\supset_\iota$ which essentially attempts to choose between x and y. But that is impossible by Lemma 7. Therefore, our pif_ι program cannot have even this more general form. □

Now we are ready to prove the main result of this section.

Proposition 9 *PCF + por cannot implement pif_ι with a constant-depth circuit.*

Proof. Assume there exists a constant-depth circuit computing pif_ι. There are two possibilities:

(i) Static circuit. The result has type integer. Therefore, there are two cases for the last gate in the circuit:
   (a) $\supset_\iota$. By Lemma 8 this is not possible.
   (b) $+1$ or -1. The circuit cannot be constructed entirely out of $+1$, -1, integers, x, y, because the result would be either hard-coded (and it must work for an infinite number of values), or a fixed offset of x or y. Also, since the circuit is static, there can only be a constant number of $+1$ or -1 in a row before reaching an occurrence of $\supset_\iota$. Then we have essentially the same situation as in case (1a) (modulo some fixed offset, as in the proof of Lemma 8), and by the same argument the circuit cannot implement pif_ι.

(ii) Dynamic circuit. We want to show that the circuit cannot be constant-depth. Assume, for a contradiction, that there is a fixed maximum constant depth beyond which the recursion does not get unwound, regardless of the inputs b, x, y. Then there are only finitely many constant-depth circuits which could be the result of the unwinding. But there are infinitely many possible inputs. Therefore, at least one of these circuits must work for infinitely many inputs. Apply the same reasoning on that circuit as in case (1) of this proof. We can assume there is no other recursion, otherwise continue the argument on the innermost recursion, which must exist because of the constant-depth assumption. Therefore, there is no fixed maximum depth for unwinding the recursion computing pif_ι.

In conclusion, it is impossible to write a constant-depth program using *por* to compute pif_ι, therefore *por* and pif_ι are not intensionally equivalent. □

4.2 Query versus pif_ι

In order to compare query to the other constructs, we need to make finer-grained distinctions than those in the previous section, and consequently, circuit semantics is no longer useful in the form we have presented it. The problem is due to the fact that we have a mixture of sequential and parallel constructs in the language and circuit semantics is an inherently parallel semantics. For a non-parallel evaluation strategy this implies that the running time of a program does not correspond closely to its circuit depth. To make the comparisons in this section we need to extend PCF's operational semantics with a notion of running time. We omit the details, but note that the results of this section only apply to evaluation strategies that are not parallel on the sequential constructs of PCF.

To see that query is more powerful consider the implementation of an n-ary function, such as n-ary addition. Assume the existence of an addition operation $(+)$, so we can write sequential addition without having to use recursion: $add_2 \equiv \lambda xy.\ x + y$. We can implement binary addition with query as follows:

$$\begin{aligned}\text{padd} \equiv \lambda x_1 x_2.\ &\text{query } (x_1, x_2) \text{ is} \\ &(v_1, v_2) \Rightarrow v_1 + v_2\end{aligned}$$

Note that the addition of v_1 and v_2 is performed sequentially (this $+$ is sequential, not bitwise-parallel). This is not essential. What is important is that separate processes are started to evaluate the inputs. Thus, we can implement n-ary addition in depth $log\ n$ by constructing a tree of binary additions.

So the question we are concerned with is whether pif_ι can also implement n-ary addition efficiently. The answer is no. The problem is that even though pif_ι can start parallel subcomputations to evaluate two integers, it must return one of them. There is no way to *combine* the results of the subcomputations. Only a limited amount of communication exists between the subcomputations: a check for equality of their results.

Proposition 10 *PCF + pif_ι cannot implement n-ary addition in depth $log\ n$.*

Proof. We identify a property that holds for our query program, *padd*, and show that it does not hold for programs of PCF + pif_ι. In *padd* the inputs are evaluated in parallel and the result is their sum. In PCF + pif_ι, the only parallel primitive is pif_ι so the inputs x and y must go through some pif_ι if they are to be evaluated in parallel. Suppose x goes through pif_ι after passing through some constant-depth circuit computing F and similarly for y and a function G. Then the output of the pif_ι is either $F(x)$ or $G(y)$. If either

$F(x) = x + y$ or $G(y) = x + y$, then the addition was performed sequentially before the pif_ι. If the output of pif_ι goes into some constant-depth H such that $H(F(x)) = x + y$ or $H(G(y)) = x + y$ then the addition was also performed sequentially, this time after the pif_ι. So it is not possible to compute $x + y$ using pif_ι in such a way that x and y are evaluated in parallel. Therefore, a PCF + pif_ι program for n-ary addition must be of depth at least n. □

As a corollary of the previous two propositions, we have the following:

Proposition 11 *PCF + por cannot implement n-ary addition in depth log n.*

In light of these results, we have the emergence of a picture of different levels of intensional expressiveness for deterministic parallel constructs: At the lowest level we have *por* and pif_o, which seem to be able to speed up only n-ary boolean functions. At the next level we have pif_ι, which can be used to speed up some integer functions. Finally, at the top level we have query, which can be used to speed up n-ary addition.

5 Comparing deterministic and nondeterministic query

A natural question to ask, after the results of the previous section, is whether relaxing the determinism constraint on the deterministic query gives us an increase in intensional expressiveness. This turns out to be a difficult question. In order to answer it we are forced to make some concessions: First, we consider a recursion-free version of PCF, since the recursive part does not mesh well with our interpretation for nondeterministic query. This is not that important, however, since we shall be making a connection with boolean circuits, which are not recursive. Second, we make a hardware assumption, in order to render the result of a subset of nondeterministic queries deterministic. This, of course, implies that we are not really comparing a deterministic and a nondeterministic construct, but rather two different machine models. We also need to go back and revisit the circuit semantics in order to obtain a precise correspondence between the dimensions of a circuit and the running time and work required to execute a program. We begin by introducing the modified language and nondeterministic query.

5.1 Recursion-free PCF

The big departure from the previous description of PCF is the lack of recursion. However, since we still need to talk about undefined (or "missing") inputs, we

introduce "undefined" constants Ω^σ, one for each type σ. We also expand somewhat the set of arithmetic constants, but this is not essential. The new set of constants we will consider is:

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

5.2 Nondeterministic query

We drop the consistency requirement on the various outputs of a query. As an example of the programs we can write now, here is one that turns out to be important in what follows:

$not_\perp \equiv \lambda x.$query (x) is
$tt \Rightarrow ff$
$ff \Rightarrow ff$
$_ \Rightarrow tt$

The semantics we presented earlier for deterministic query (Figure 3) also makes sense for nondeterministic query, but now permits the amb operator to be presented with distinct defined inputs. Under this interpretation, however, programs no longer compute functions, but relations. Given the fact that our definition of intensional expressiveness requires the computation of functions, we would have to restrict the indeterminacy to the inside of a program. The following proposition shows that we cannot do that in any useful way.

Proposition 12 *Under the semantics of Figure 3, nondeterministic query is not intensionally more expressive than deterministic query.*

Proof. Let P be a nondeterministic program which computes the function f, and suppose it uses $amb(x, y)$, with $x \neq y$. Since P must compute a deterministic answer (by the definition of intensional expressiveness), the $amb(x, y)$ must be "determinized" somehow. Essentially, the only way that could be achieved is to throw it out: either not use it, or use it in one branch of a conditional that always chooses the other branch, or use it as the argument to a function whose result is independent of its input. But then we can certainly write a deterministic version of P that computes f with the same efficiency. □

We still want to restrict ourselves to programs that compute functions, even though they may use nondeterministic query, but we need a different inter-

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

$$\text{exhausted}(x^o, i, j, p_1, \ldots, p_k, \rho) = \begin{cases} tt, \text{ if } \begin{array}{l} ((\exists l, m.\ \mathcal{D}[\![e_{lj}]\!]\rho = tt \wedge \mathcal{D}[\![e_{mj}]\!]\rho = ff) \vee \\ \quad (\exists l.\ \mathcal{D}[\![e_{lj}]\!]\rho = y^o)) \wedge (p_1 \backslash j \Uparrow \cdots \Uparrow p_k \backslash j) \end{array} \\ ff, \text{ otherwise} \end{cases}$$

$$\text{exhausted}(x^\iota, i, j, p_1, \ldots, p_k, \rho) = \begin{cases} tt, \text{ if } (\exists l.\ \mathcal{D}[\![e_{lj}]\!]\rho = y^\iota) \wedge (p_1 \backslash j \Uparrow \cdots \Uparrow p_k \backslash j) \\ ff, \text{ otherwise} \end{cases}$$

Fig. 8. Semantics for nondeterministic query

pretation for the meaning of nondeterministic query. One possibility is shown in Figure 8. The idea is to allow the "don't care" symbol to represent $\bot$ in certain circumstances, e.g., when the corresponding pattern position has been exhaustively checked for all other alternatives, and the remainder of the pattern is consistent. The semantics is the same as in Figure 3 except for the "don't care" symbol. Let us say that such a symbol is found at location j in the i^{th} pattern (written $__{ij}$). If the other patterns exhaustively check the input at location j and are otherwise consistent ($p\backslash j$ refers to the pattern p without location j), then the meaning of matching x^σ to "$_$" is a *poll* of the input. *Poll* is a nondeterministic construct [25] which checks whether an input is available:

$$poll\ \bot = ff, \qquad poll\ x = tt, \text{ if } x \neq \bot.$$

Now we can identify a subset of the nondeterministic queries which return deterministic results, assuming we can detect undefined inputs. When we have a "$__{ij}$" interpreted as $\bot$ we can take its meaning to be $\mathcal{D}[\![__{ij}]\!]\rho = \star$, where we have added the element $\star$ to the flat domains D_{bool}, D_{int}, with $\bot \sqsubseteq \star$. Then under the above definition of consistency we obtain the desired queries.

We call the extension of PCF with deterministic query DPCF, and the extension with nondeterministic query NPCF. It is quite obvious that NPCF is extensionally more expressive than DPCF, but we are interested in the following question: *Is NPCF intensionally more expressive than DPCF?* Since NPCF is an extension of DPCF it can certainly compute as efficiently as DPCF. However, to show that NPCF is more expressive, we must exhibit a function expressible in both DPCF and NPCF, and prove that DPCF cannot compute it as efficiently. This question turns out to be analogous to a problem in computational complexity theory, that of comparing monotone and De Morgan boolean circuits. Before showing why, we return to our circuit semantics.

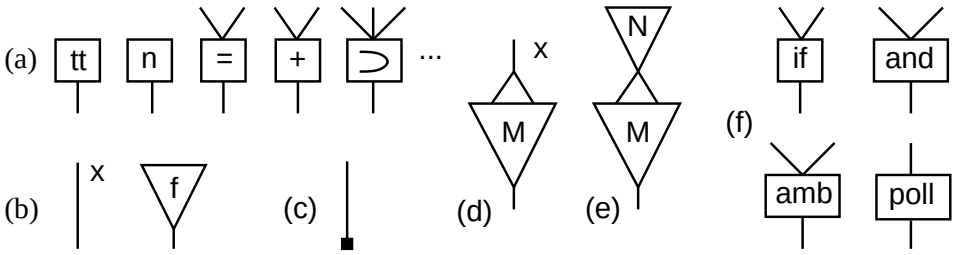

Fig. 9. Circuits for (a) constants, (b) variables, (c) ignored inputs, (d) abstraction, (e) application, (f) query

6 Circuit semantics revisited

As we have seen, the basic idea of circuit semantics is very simple and has much in common with dataflow networks: view each construct of PCF as a gate, and view a computation as data flowing through the gate. The whole program becomes a circuit. Earlier we considered the circuits as being executed bottom-up. This reflected our intuition about recursion being unwound on demand. But now we wish to view them as being executed top-down, given our upcoming comparison with boolean circuits. We shall also make very precise the relationship between the dimensions of a circuit and various parallel evaluation orders for PCF, and introduce new gates to model queries.

6.1 Circuits for PCF

Figure 9(a)(b) shows the circuits representing constants and variables. The truth values and the integers are represented by nodes with no inputs. Nodes may have several outputs (fan-out is unlimited). Each of the functional constants is a node with the required number of inputs. A variable is represented by a wire, or for higher-order functions, a placeholder circuit labelled with the variable name. Figure 9(c) shows an input that is ignored; we need such a convention to write functions like the K combinator. The representation of conditionals in Figure 9(a) shows one of the essential differences with dataflow networks, which use switch and merge nodes to avoid evaluating more than one branch of the conditional during *data-driven* (top-down) execution. For *demand-driven* (bottom-up) execution of the network the difference is irrelevant. Figure 9(d)(e) shows the representation of lambda abstraction and application. This points out the other major difference with dataflow networks: there are no application nodes. Figure 10 shows the circuits representing three simple PCF terms. Note that our representation builds in sharing of arguments of ground type (but functional arguments are not shared).

Since it is rather difficult to reason about circuit dimensions in pictorial form, we define a model for the dimensions of a circuit. Figure 11 is an extended

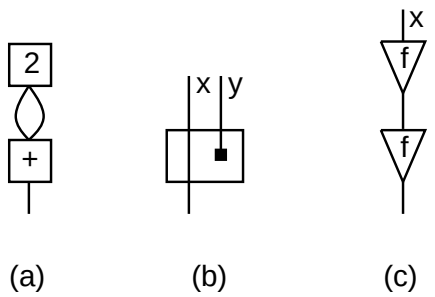

Fig. 10. Examples: (a) $(\lambda x.\ x + x)2$, (b) $\lambda xy.\ x$, (c) $\lambda fx.\ f(fx)$

semantics for PCF, returning step-counting versions of a term, *S-Terms* (*cf. t-programs* in [29]), computing its depth and size. Evaluating the step-counting depth and size translations of a program gives us its depth and size. We assume renaming of bound variables, so that all identifiers are unique. The η environment in the size translation ensures sharing of arguments of ground type. π_1 is the first projection. The syntax of S-Terms uses Λ rather than λ merely to emphasize the distinction between the programming language and the meta-language.

Example 13 Consider the program $(\lambda x.\ x + x)2$, whose circuit semantics is depicted in Figure 10(a). The meaning of the program under the extended semantics is:

$$\mathcal{E}[\![(\lambda x.\ x + x)2]\!]\bot = (4, (\Lambda x.\ 1 + \max(x, x))1, (\Lambda x.\ 1 + x + 0)1).$$

Evaluating the depth and size step-counting programs, we get a depth and size of 2, which match the circuit dimensions in Figure 10(a). □

We now prove that the circuit dimensions are indeed matched by the extended semantics.

Proposition 14 *For a PCF program M, $\mathcal{E}[\![M]\!]\bot = (v, d, s)$ if and only if the circuit representing M has depth d and size s.*

Proof. By induction on the structure of M. We need an induction hypothesis that works at higher types (*cf.* [26] for a similar example). We define predicates Match^σ by induction on types:

(i) If M^σ is a program, then M^σ has property Match^σ if $\mathcal{E}[\![M^\sigma]\!]\bot = (v, d, s)$ if and only if the circuit representing M has depth d and size s.
(ii) If $M^{\sigma\to\tau}$ is a closed term, then it has property $\text{Match}^{\sigma\to\tau}$ if whenever N^σ is a closed term with property Match^σ, $M^{\sigma\to\tau}\ N^\sigma$ has property Match^τ.
(iii) If M^σ is an open term with free variables $x_1^{\sigma_1}, \ldots, x_n^{\sigma_n}$, then it has property Match^σ if the term $[N_1/x_1]\cdots[N_n/x_n]M^\sigma$ has property Match^σ whenever

$\mathcal{E}$: Terms $\to$ Environments $\to$($\bigcup D_\sigma \times$ S-Terms $\times$ S-Terms)

$\mathcal{E}[\![M]\!]\rho = (\mathcal{D}[\![M]\!]\rho, depth(M), \pi_1(size(M)\bot))$, where

$depth$: Terms $\to$ S-Terms

$depth(tt) = 1$

$depth(n) = 1$

$depth(M_1 = M_2) = 1 + \max(depth(M_1), depth(M_2))$

$depth(M_1 + M_2) = 1 + \max(depth(M_1), depth(M_2))$

$depth(\supset_\iota\ M_1\ M_2\ M_3) = 1 + \max(depth(M_1), depth(M_2), depth(M_3))$

$depth(x^\sigma) = x^\sigma$

$depth(M_1\ M_2) = depth(M_1)\ depth(M_2)$

$depth(\lambda x^\sigma.\ M) = \Lambda x^\sigma.\ depth(M)$

$size$: Terms $\to$ Environments $\to$(S-Terms $\times$ Environments)

$size(tt)\eta = (1, \eta)$

$size(n)\eta = (1, \eta)$

$size(M_1 = M_2)\eta = (1 + s_1 + s_2, \eta'')$, where $(s_1, \eta') = size(M_1)\eta$, $(s_2, \eta'') = size(M_2)\eta'$

$size(M_1 + M_2)\eta = (1 + s_1 + s_2, \eta'')$, where $(s_1, \eta') = size(M_1)\eta$, $(s_2, \eta'') = size(M_2)\eta'$

$size(\supset_\iota\ M_1\ M_2\ M_3)\eta = (1 + s_1 + s_2 + s_3, \eta''')$, where

$$(s_1, \eta') = size(M_1)\eta,\ (s_2, \eta'') = size(M_2)\eta',\ (s_3, \eta''') = size(M_2)\eta''$$

$$size(x^\sigma)\eta = \begin{cases} (x^\sigma, \eta[x^\sigma \mapsto 1]), & \text{if } \sigma \in \{o, \iota\} \text{ and } \eta(x^\sigma) = \bot \\ (0, \eta), & \text{if } \sigma \in \{o, \iota\} \text{ and } \eta(x^\sigma) = 1 \\ (x^\sigma, \eta), & \text{otherwise} \end{cases}$$

$size(M_1\ M_2)\eta = (s_1 s_2, \eta'')$, where $(s_1, \eta') = size(M_1)\eta$, $(s_2, \eta'') = size(M_2)\eta'$

$size(\lambda x^\sigma.\ M)\eta = (\Lambda x^\sigma.\ s_1, \eta')$, where $(s_1, \eta') = size(M)\eta$

Fig. 11. Extended semantics for PCF

$N_1, \ldots, N_n$ are closed terms having properties $\text{Match}^{\sigma_1}, \ldots, \text{Match}^{\sigma_n}$, respectively.

We call a term M^σ *matching* if it has property Match^σ. We need to prove that all terms are matching. This is obvious for boolean and integer constants. We only consider one functional constant, as the proof is similar for the others.

(i) $M \equiv M_1 + M_2$. By induction hypothesis, M_1, M_2 have property Match^ι. We have several cases depending on whether the terms M_1, M_2 are closed or open:
   (a) M_1, M_2 are both closed. Then by case (1) of the definition of Match^ι, $\mathcal{E}[\![M_1]\!]\perp = (v_1, d_1, s_1)$ and $\mathcal{E}[\![M_2]\!]\perp = (v_2, d_2, s_2)$ iff the circuits representing M_1, M_2 have dimensions d_1, s_1 and d_2, s_2, respectively. Since both terms are closed there is no possible sharing, so by the definition of $\mathcal{E}$ and of the circuits, the depth and size of $M_1 + M_2$ are $1 + \max(d_1, d_2)$, $1 + s_1 + s_2$, respectively.
   (b) M_1 is open, M_2 is closed. By case (3) of the definition of Match^ι, any closed instantiation of M_1 with matching terms will have property Match^ι. Since M_2 is closed, there is no sharing between M_1 and M_2, so, as before, the dimensions match.
   (c) M_1, M_2 are both open. Consider the set $S = FV(M_1) \cap FV(M_2)$, restricted to variables of ground type. By case (3) of the the induction hypothesis, any closed instantiation of M_1, M_2 with matching terms will have property Match^ι. Then the depth of $M_1 + M_2$ will be the same in the two models, since sharing is not relevant. The size of $M_1 + M_2$ in the circuit model will only count each instantiation of a shared variable $x^\sigma \in S$ once. But that is exactly what the η environment is for in the $\mathcal{E}$ model. Therefore, the size will also match.

(ii) $M \equiv x^\sigma$. Any closed instantiation of x^σ by a term satisfying Match^σ will have the same property.

(iii) $M \equiv M_1^{\sigma \to \tau} M_2^\sigma$. By the induction hypothesis, $M_1^{\sigma \to \tau}$ satisfies property $\text{Match}^{\sigma \to \tau}$ and M_2^σ has property Match^σ. If both M_1, M_2 are closed, then by case (2) of the induction hypothesis, $M_1^{\sigma \to \tau} M_2^\sigma$ has property Match^τ. If M_1, M_2 are open, then construct the set S as above. Any closed instantiation of M_1, M_2 with matching terms will result in the sharing of $x \in S$, both in circuits and in the $\mathcal{E}$ model. Therefore $M_1^{\sigma \to \tau} M_2^\sigma$ will have property Match^τ.

(iv) $M \equiv \lambda x^\sigma.\ M^\tau$. By the induction hypothesis, M^τ has property Match^τ. We need to show that $\lambda x^\sigma.\ M^\tau$ has property $\text{Match}^{\sigma \to \tau}$. Suppose that $\lambda x^\sigma.\ M^\tau$ is closed. If $\sigma \in \{o, \iota\}$, then for any input N^σ both the circuits and the $\mathcal{E}$ model will only include one copy of N^σ. The depth function from the $\mathcal{E}$ model actually does no sharing, but this is irrelevant, as the depth is unaffected. Then $(\lambda x^\sigma.\ M^\tau)N^\sigma$ will have property Match^τ. For general σ, there are no sharing issues, so the result again has property Match^τ. Now suppose $\lambda x^\sigma.\ M^\tau$ is open. Any closed instantiation of

$\lambda x^\sigma.\ M^\tau$ with matching terms will result in the same sharing of common variables in both circuits and the $\mathcal{E}$ model. Therefore, $\lambda x^\sigma.\ M^\tau$ has property $\mathrm{Match}^{\sigma \to \tau}$.

Since all M^σ have property Match^σ, certainly the programs enjoy this property as well, which by case (1) of the definition of Match^σ establishes our proposition. $\square$

Before undertaking a comparison of our circuit model with various evaluation strategies, we discuss the issues involved. Consider the following PCF program: $P \equiv \supset_\iota\ tt\ 2\ M$, where M is an expression whose evaluation takes many steps before returning an integer. The circuit representation of P will include a piece corresponding to M. But in call-by-name PCF (or call-by-need, a particular implementation of call-by-name), M will not be evaluated, so the size of the circuit representing P will be a wild overestimate. Of course, we could evaluate the circuit bottom-up, thus avoiding evaluation of M, but this would preclude any meaningful discussion of circuit size or depth. The problem is that circuits are an inherently parallel model, and call-by-name (or call-by-need) is an inherently sequential evaluation strategy. Thus it seems reasonable to confine ourselves to evaluation strategies which can be usefully parallelized.

We compare our circuit model to two different evaluation strategies: parallel call-by-value (c-b-v) [19,3] and call-by-speculation (c-b-s) [19,28,16]. Given an application $f\ x$, in parallel c-b-v the function f and the argument x are evaluated in parallel and after *both* evaluations are completed, then the body of the function f is evaluated. Therefore, parallel c-b-v takes advantage of *horizontal parallelism*, and also to a lesser extent of *vertical parallelism*.

In c-b-s, we also evaluate f and x in parallel, but we do not require that the evaluation of x be complete before we evaluate the body of f. Thus c-b-s allows fully pipelined parallelism. If x gives rise to a large computation that is not used by f we will get a result much faster, but the computation will still continue for some time. C-b-s thus introduces a distinction between *minimum* and *maximum* time to evaluate an expression.

Figure 12 shows a profiling semantics for parallel c-b-v in the style of [3]. Judgments have the form $\rho \vdash M \longrightarrow_{\mathrm{cbv}} v; d, w$, meaning that in environment ρ, M evaluates to v in d steps (depth) and with w work. The possible results of an evaluation are values, ranged over by v, and they are either constants or function closures:

$v\ ::= c \mid cl(\rho, x, M)$.

The constants chosen for the depth and work in several of the rules are different from [3], as we want to achieve an exact match with our circuit semantics.

$$\rho \vdash tt \longrightarrow_{\mathrm{cbv}} tt; 1, 1$$

$$\frac{\rho \vdash M_1 \longrightarrow_{\mathrm{cbv}} v_1; d_1, w_1 \quad \rho \vdash M_2 \longrightarrow_{\mathrm{cbv}} v_2; d_2, w_2}{\rho \vdash M_1 + M_2 \longrightarrow_{\mathrm{cbv}} v_1 + v_2; 1 + \max(d_1, d_2), 1 + w_1 + w_2}$$

$$\frac{\rho \vdash M_1 \longrightarrow_{\mathrm{cbv}} tt; d_1, w_1 \quad \rho \vdash M_2 \longrightarrow_{\mathrm{cbv}} v_2; d_2, w_2}{\rho \vdash \supset_\iota M_1 M_2 M_3 \longrightarrow_{\mathrm{cbv}} v_2; 1 + d_1 + d_2, 1 + w_1 + w_2}$$

$$\frac{\rho \vdash M_1 \longrightarrow_{\mathrm{cbv}} \mathit{ff}; d_1, w_1 \quad \rho \vdash M_3 \longrightarrow_{\mathrm{cbv}} v_3; d_3, w_3}{\rho \vdash \supset_\iota M_1 M_2 M_3 \longrightarrow_{\mathrm{cbv}} v_3; 1 + d_1 + d_3, 1 + w_1 + w_3}$$

$$\frac{\rho(x) = v}{\rho \vdash x \longrightarrow_{\mathrm{cbv}} v; 0, 0}$$

$$\rho \vdash \lambda x.\, M \longrightarrow_{\mathrm{cbv}} cl(\rho, x, M); 0, 0$$

$$\frac{\begin{array}{l}\rho \vdash M_1 \longrightarrow_{\mathrm{cbv}} cl(\rho', x, M_1'); d_1, w_1 \\ \rho \vdash M_2 \longrightarrow_{\mathrm{cbv}} v_2; d_2, w_2\end{array} \qquad \rho'[x/v_2] \vdash M_1' \longrightarrow_{\mathrm{cbv}} v; d_3, w_3}{\rho \vdash M_1\ M_2 \longrightarrow_{\mathrm{cbv}} v; \max(d_1, d_2) + d_3, w_1 + w_2 + w_3}$$

Fig. 12. Profiling semantics for parallel call-by-value

The correspondence between our circuit semantics and parallel c-b-v is fairly simple: as long as we have conditionals in a program, the models are incomparable. However, without conditionals, circuits will take less time and do less work than predicted by the operational semantics, in the following sense:

Proposition 15 *Given a PCF program M with $\mathcal{E}[\![M]\!]\bot = (v, d, s)$, and $\bot \vdash M \longrightarrow_{\mathrm{cbv}} v; t, w$, the following hold:*

(i) If M is conditional-free, then $s \leq w$;
(ii) If M is conditional-free, then $d \leq t$.

Proof. By induction on M as in the proof of Proposition 14. We define predicates Val^σ by induction on types:

(i) If M^σ is a program, then M^σ has property Val^σ if $\mathcal{E}[\![M^\sigma]\!]\bot = (v, d, s)$ if and only if $\bot \vdash M \longrightarrow_{\mathrm{cbv}} v; t, w$, with $s \leq w, d \leq t$.

(ii) If $M^{\sigma\to\tau}$ is a closed term, then it has property $\mathrm{Val}^{\sigma\to\tau}$ if whenever N^σ is a closed term with property Val^σ, $M^{\sigma\to\tau}\ N^\sigma$ has property Val^τ.

(iii) If M^σ is an open term with free variables $x_1^{\sigma_1},\ldots,x_n^{\sigma_n}$, then it has property Val^σ if the instantiation $[N_1/x_1]\cdots[N_n/x_n]M^\sigma$ has property Val^σ whenever $N_1,\ldots,N_n$ are closed terms having properties $\mathrm{Val}^{\sigma_1},\ldots,\mathrm{Val}^{\sigma_n}$, respectively.

The result is easily verifiable for constants (except, of course, for conditionals), so we only consider the induction step:

(i) $M\equiv x^\sigma$. Any closed instantiation of x^σ by a term satisfying Val^σ will have the same property.

(ii) $M\equiv M_1^{\sigma\to\tau}M_2^\sigma$. By the induction hypothesis, $M_1^{\sigma\to\tau}$ has property $\mathrm{Val}^{\sigma\to\tau}$ and M_2^σ has property Val^σ. If both M_1, M_2 are closed, then by case (2) of the induction hypothesis, $M_1^{\sigma\to\tau}M_2^\sigma$ has property Val^τ. If M_1, M_2 are open, then construct the set $S=FV(M_1)\cap FV(M_2)$, restricted to variables of ground type. Any closed instantiation of M_1, M_2 with closed terms satisfying Val will result in the sharing of $x^{\in S}$, in both circuits and parallel c-b-v. Therefore $M_1^{\sigma\to\tau}M_2^\sigma$ will have property Val^τ.

(iii) $M\equiv\lambda x^\sigma.\ M^\tau$. We need to show that $\lambda x^\sigma.\ M^\tau$ has property $\mathrm{Val}^{\sigma\to\tau}$. By the induction hypothesis, M^τ has property Val^τ. Suppose that $\lambda x^\sigma.\ M^\tau$ is closed. Let N^σ be a closed term satisfying Val^σ. If M^τ is closed (*i.e.,* x^σ is not used in M^τ), then $depth((\lambda x^\sigma.\ M^\tau)N^\sigma)=depth(M^\tau)$ and similarly for size. Since M^τ has property Val^τ, so must $(\lambda x^\sigma.\ M^\tau)N^\sigma$ (parallel c-b-v requires more time and work to evaluate $(\lambda x^\sigma.\ M^\tau)N^\sigma$ than M^τ). This is the origin of the inequalities in the proposition: parallel c-b-v will evaluate unused arguments.

If M^τ is open, then if $\sigma\in\{o,\iota\}$, N^σ will be shared in both circuits and parallel c-b-v, so the depth and work will be the same. For general σ, by the third part of the induction hypothesis, $[N^\sigma/x^\sigma]M^\tau$ will have property Val^τ.

Now suppose $\lambda x^\sigma.\ M^\tau$ is open. Any closed instantiation of $\lambda x^\sigma.\ M^\tau$ with closed terms satisfying Val will result in the same sharing of common variables of ground type in both circuits and parallel c-b-v. Therefore, $\lambda x^\sigma.\ M^\tau$ has property $\mathrm{Val}^{\sigma\to\tau}$.

Since all terms M^σ have property Val^σ, so do the programs, which establishes our result. □

Figure 13 shows a profiling semantics for c-b-s, in the style of [16]. Because of the distinction between minimum and maximum depth, the judgments are different from the parallel c-b-v ones. They are of the form

$$\rho, d \vdash M \longrightarrow_{\mathrm{cbs}} v; d', \hat{d}', w,$$

$$\rho, d \vdash tt \longrightarrow_{\mathrm{cbs}} tt; d+1, d+1, 1$$

$$\frac{\rho, d \vdash M_1 \longrightarrow_{\mathrm{cbs}} v_1; d_1, \hat{d}_1, w_1 \quad \rho, d \vdash M_2 \longrightarrow_{\mathrm{cbs}} v_2; d_2, \hat{d}_2, w_2}{\rho, d \vdash M_1 + M_2 \longrightarrow_{\mathrm{cbs}} v_1 + v_2; 1 + \max(d_1, d_2), 1 + \max(\hat{d}_1, \hat{d}_2), 1 + w_1 + w_2}$$

$$\frac{\begin{array}{l}\rho, d \vdash M_1 \longrightarrow_{\mathrm{cbs}} tt; d_1, \hat{d}_1, w_1 \\ \rho, d \vdash M_2 \longrightarrow_{\mathrm{cbs}} v_2; d_2, \hat{d}_2, w_2\end{array} \quad \rho, d \vdash M_3 \longrightarrow_{\mathrm{cbs}} v_3; d_3, \hat{d}_3, w_3}{\rho, d \vdash \supset_\iota M_1 M_2 M_3 \longrightarrow_{\mathrm{cbs}} v_2; 1 + \max(d_1, d_2), 1 + \max(\hat{d}_1, \hat{d}_2, \hat{d}_3), 1 + w_1 + w_2 + w_3}$$

$$\frac{\begin{array}{l}\rho, d \vdash M_1 \longrightarrow_{\mathrm{cbs}} ff; d_1, \hat{d}_1, w_1 \\ \rho, d \vdash M_2 \longrightarrow_{\mathrm{cbs}} v_2; d_2, \hat{d}_2, w_2\end{array} \quad \rho, d \vdash M_3 \longrightarrow_{\mathrm{cbs}} v_3; d_3, \hat{d}_3, w_3}{\rho, d \vdash \supset_\iota M_1 M_2 M_3 \longrightarrow_{\mathrm{cbs}} v_3; 1 + \max(d_1, d_3), 1 + \max(\hat{d}_1, \hat{d}_2, \hat{d}_3), 1 + w_1 + w_2 + w_3}$$

$$\frac{\rho(x) = v; d'}{\rho, d \vdash x \longrightarrow_{\mathrm{cbs}} v; \max(d, d'), \max(d, d'), 0}$$

$$\rho, d \vdash \lambda x.\, M \longrightarrow_{\mathrm{cbs}} cl(\rho, x, M); d, d, 0$$

$$\frac{\begin{array}{l}\rho, d \vdash M_1 \longrightarrow_{\mathrm{cbs}} cl(\rho', x, M_1'); d_1, \hat{d}_1, w_1 \\ \rho, d \vdash M_2 \longrightarrow_{\mathrm{cbs}} v_2; d_2, \hat{d}_2, w_2\end{array} \quad \rho'[x/v_2; d2], d_1 \vdash M_1' \longrightarrow_{\mathrm{cbs}} v; d_3, \hat{d}_3, w_3}{\rho, d \vdash M_1\ M_2 \longrightarrow_{\mathrm{cbs}} v; d_3, \max(\hat{d}_1, \hat{d}_2, \hat{d}_3), w_1 + w_2 + w_3}$$

Fig. 13. Profiling semantics for call-by-speculation

meaning that in environment ρ, evaluating M at depth d leads to result v which will be available at depth d', while the whole computation will be done at depth $\hat{d}'$ and use w work. In addition, the environment ρ has to keep track at which depth a value becomes available. This is equivalent to the effect achieved in Roe's semantics [28] with time stamps.

It should be fairly clear by now that the circuit semantics is very closely related to c-b-s. In fact, it would be a slight improvement on it were it not for conditionals. A circuit will always do less work than c-b-s, but its depth might be larger than the c-b-s minimum depth. In a conditional-free program P, however, the depth of the circuit semantics of P is the same as the c-b-s

minimum depth.

Proposition 16 *If* $\mathcal{E}[\![M]\!]\perp = (v, d, s)$, *and* $\perp, 0 \vdash M \longrightarrow_{\text{cbs}} v'; t, \hat{t}, w$, *then* $v = v'$ *and we have: (1)* $s \leq w$; *(2)* $d \leq \hat{t}$; *(3) if* M *is conditional-free, then* $d = t$.

Proof. Similar to the proof of Proposition 15. □

There is another order of evaluation to which we could compare our circuit model. Hudak and Anderson [19] consider parallel *eager* evaluation, a version of parallel c-b-v. Instead of requiring the argument to complete evaluating before the body of the function is evaluated, we require that it complete before the function call returns. This gives extra parallelism over parallel c-b-v, but might still evaluate some arguments unnecessarily. The correspondence between parallel eager evaluation and circuits would be the same as that between parallel c-b-v and circuits.

6.2 Circuits for query

The circuit semantics of query is a straightforward implementation of the semantics presented earlier. Accordingly, we need to expand our set of basic gates with those of Figure 9(f). An "if-then" gate is just shorthand for an if-then-else statement, with a $\perp$ "else" part. *And* is parallel-and. For space reasons *amb* is depicted at times with more than two inputs; the understanding is that a k-ary *amb* gate stands for a balanced tree of binary *amb* gates.

Figure 14(a) shows the circuit semantics of parallel-or. The *not* gate is shorthand for $\supset_o\ x\ ff\ tt$. Figure 14(b) shows the circuit semantics of $not_\perp$ from Section 5.2, which essentially performs negation with respect to $\perp$ (note how "_" is used to represent $\perp$).

7 On the method and the metric

This section describes our methodology for comparing DPCF and NPCF. First, we refine the notion of intensional expressiveness to take into account both measures of program complexity induced by the circuit semantics: work and time. Then we compare our circuit model with the traditional parallel computation model, the PRAM, establishing a simple connection via an analogue of Brent's theorem. Finally, we detail the hardware assumption we make that allows us to compare nondeterministic and deterministic programs.

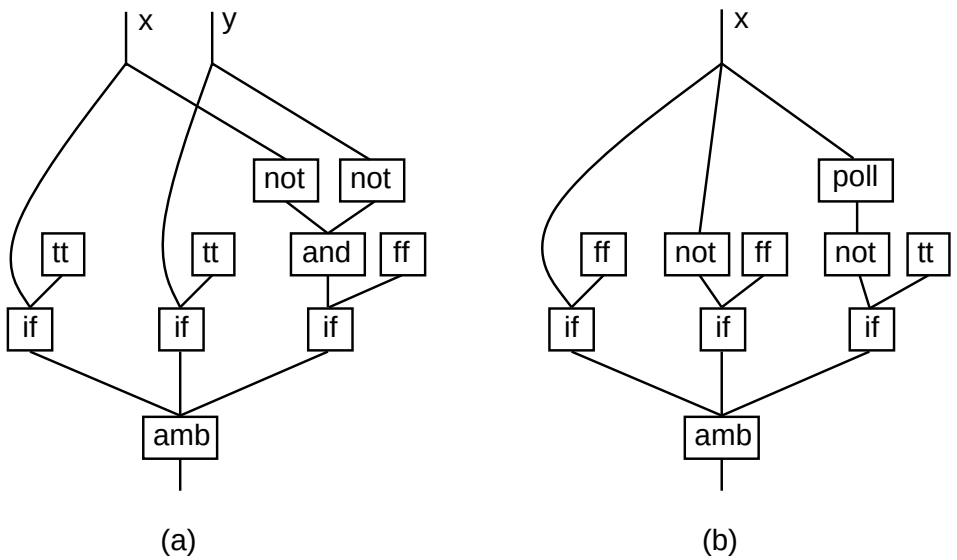

Fig. 14. Circuit semantics for: (a) *por*, (b) $not_\perp$

7.1 Intensional expressiveness and parallel complexity

Given a program P, the circuit semantics of P provides a definition of its parallel complexity: the parallel time and the parallel work required to execute it. Accordingly, we can define two notions of intensional expressiveness.

We say that language L_1 is *intensionally more work-expressive* than L_2 ($L_1 \succeq_w L_2$), if any function computable by L_2 with a program whose size under the circuit semantics is s_2 can be computed by an L_1 program with circuit size $s_1 \leq s_2$. Similarly, L_1 is *intensionally more time-expressive* than L_2 ($L_1 \succeq_t L_2$), if any function computable by L_2 with a program whose depth under the circuit semantics is d_2 can be computed by an L_1 program with circuit depth $d_1 \leq d_2$.

As mentioned in the introduction, we are interested in asymptotic complexity: for a function f of n arguments, we want to compare the size and depth of circuits computing f as functions of n. In particular, we would like to establish separation results of the type $L_1 \succ_w L_2$ and $L_1 \succ_t L_2$.

7.2 Comparison with the PRAM

We compare our parallel complexity model, circuit semantics, with the standard model from the theory of parallel algorithms, the PRAM [10]. This comparison will be needed later when we establish a connection between PCF programs and boolean circuits.

Suppose we have a program M whose parallel complexity according to the

circuit semantics is size s and depth d. The question we have to address is how long it takes to execute M on a p-processor PRAM.

We can imagine the process of executing M in two parts: construction and execution. First we construct the circuit semantics of M, then we execute it. We consider the execution part first, since it is very simple.

Proposition 17 *[Execution] Given the circuit semantics of program M, of size s and depth d, it can be simulated on a p-processor CREW PRAM in $O(s/p + d)$.*

Proof. This follows from Brent's theorem [10], which establishes bounds for simulating boolean circuits on the PRAM. Our circuits satisfy the necessary conditions for Brent's theorem: we have bounded fan-in, and we assume that each gate can be simulated in $O(1)$ time. □

For our purposes, we do not need a Construction proposition. Rather, we want to know if there is any difference in construction time between DPCF and NPCF programs. Since the only difference is in the treatment of queries, we only consider the time needed to generate circuits for each of the two kinds of queries.

Proposition 18 *[Construction equivalence] It takes asymptotically the same time on a p-processor PRAM to construct circuits for deterministic and nondeterministic interpretations of a query.*

Proof. To implement deterministic query we need to look at every element of each pattern at least once. Then the time required to process a query on n variables and containing k patterns be must be at least $O(nk/p)$.

For the nondeterministic interpretation, we have to check if any "_" is supposed to be $\perp$. This can be achieved as follows: First check if any column in the pattern exhaustively checks for all possibilities, and identify all "_" that occur in those columns. Also, keep track if the column consists entirely of variables and "_". This can be done in $O(nk/p)$.

If all columns contain only variables and "_", then all "_" from exhaustively checked columns are $\perp$, and we are done. Otherwise, pick one column, call it i, that is exhaustively checked with more than variables and "_", and verify the rest of the pattern for consistency. This also takes $O(nk/p)$.

If consistent, then all "_" from column i are $\perp$, and we are done, since including column i in any future consistency check will fail. If not consistent, then

again we are done for the same reason.

Since we only take $O(nk/p)$ to find out which "_" mean $\perp$, it takes the same time to generate circuits for both deterministic and nondeterministic query. □

7.3 How to compare determinism and nondeterminism

Our notion of intensional expressiveness involves comparing the complexity of computing *functions*, so we have to make sure our nondeterministic programs return deterministic values. Since a purely nondeterministic interpretation of query fails to be intensionally more expressive (Proposition 12), we chose an interpretation that allows us to get deterministic answers, assuming we can detect undefined inputs. This section discusses our assumption in more detail.

There has been a considerable amount of research in the area of algorithms for unreliable distributed systems. For instance, a whole field is devoted to consensus problems in such systems [14]. In such algorithms, a key assumption is that one can detect the failure of a process to send a message. Without this assumption, *i.e.,* in a fully asynchronous model, the consensus problem is not solvable. The physical realization of this assumption is quite reasonable, and does not even require fully synchronous hardware. Lamport [23] has shown how to detect failure to send messages through the use of timeouts. This requires a model with accurate clocks and bounds on message transit times.

We shall assume that we are able to detect undefined inputs through the use of such hardware. Since DPCF cannot take advantage of this (by the definition of deterministic query), in a sense, our question has become: Does the ability to detect undefined inputs in NPCF imply that NPCF $\succ_w$ DPCF and/or NPCF $\succ_t$ DPCF?

The difficulty in answering this question stems from the fact that we have to exhibit a function computable by *both* NPCF and DPCF, but which can be computed faster or with less work by NPCF. It is not enough to say that NPCF can compute generalized functions because they have no deterministic counterpart. On the other hand, DPCF is quite powerful. It can express parallel-or and parallel-and, so it can implement comparators for boolean values which work with undefined inputs. Using comparators we can implement asymptotically very efficient sorting networks, such as the AKS network [1], which can sort n inputs in depth $O(\lg n)$ and size $O(n \lg n)$. Since a DPCF program must output a single value we cannot just implement the sorting network, but we can use it internally, for instance, to compute the majority function, by first sorting the input and then picking the middle value [34].

Such considerations led us to examine more closely the similarity between DPCF and NPCF programs and boolean circuits. It turns out that there is a very close correspondence.

8 A connection with boolean circuits

This section contains a formal connection between our DPCF and NPCF languages and boolean circuits. Since there are different kinds of boolean circuits in the literature, depending on the basis, we first give a very brief overview of the circuits we are interested in.

8.1 Boolean circuits

A boolean circuit computes a boolean function $f : \{0,1\}^n \to \{0,1\}$. For our purposes, a boolean circuit is a directed acyclic graph whose inputs are sources of the graph, and whose nodes (gates) are selected from a set called the basis. The fan-out of the inputs and gates in the circuit is unbounded. The fan-in is 2. Normally, a circuit can have more than one output, but we are only interested in single-output circuits, since our DPCF and NPCF programs only return one output.

Gates in the basis are single output. Among the many bases studied, two are of interest to us: the *monotone* basis $\{\wedge, \vee\}$, and the De Morgan basis $\{\wedge, \vee, \neg\}$. The monotone basis is not complete, i.e. not every boolean function is computable by monotone circuits (only the monotone functions are), but it is of particular interest to researchers in complexity theory, since strong lower bounds have been obtained for monotone circuits [34]. The De Morgan basis is complete.

There are two measures for the efficiency of a circuit. The *size* or *complexity* is the number of gates in the circuit; this intuitively measures the work needed to compute the output. The *depth* is the number of gates in the circuit on the longest path from the input to any output; this measures the time required to compute the output.

There are several complexity classes defined in terms of circuits. We define those which will be used in what follows (*cf.* [4]). A *family* of circuits is a sequence $(C_1, C_2, \ldots)$, where C_n takes n input variables. A *uniform family* is one where the description of C_n can easily be computed from n. The classes NC^k (AC^k), for $k \geq 0$, consist of all functions computable by a uniform family of polynomial size, $O(\log^k n)$ depth circuits with constant (unbounded) fan-in.

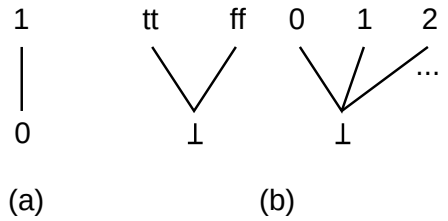

Fig. 15. Domains for (a) boolean circuits, (b) DPCF and NPCF programs

$\mathrm{NC} = \bigcup_{k \geq 0} \mathrm{NC}^k$, and similarly $\mathrm{AC} = \bigcup_{k \geq 0} \mathrm{AC}^k$. It is easy to show that for all $k \geq 0$, $\mathrm{AC}^k \subseteq \mathrm{NC}^{k+1} \subseteq \mathrm{AC}^{k+1}$, therefore $\mathrm{NC} = \mathrm{AC}$.

8.2 The connection

The main idea of the connection is very simple and can best be described by a picture. Figure 15 shows the domain of booleans for circuits, and the domains of booleans and natural numbers for DPCF and NPCF. Booleans are ordered differently in the circuit world than in programming language semantics. Monotone means the same thing in both places, but with respect to a different ordering. That is why negation is not monotone in the case of circuits, but it certainly is expressible in PCF, which only computes continuous (hence monotone) functions. Therefore, 0 has the same rôle in the circuit world that $\perp$ has for PCF programs.

When a monotone circuit is presented with an input, there is no way for the circuit to determine if that input is 0, since all it can do is AND and OR the inputs. This is really the same situation as when a DPCF program can have undefined inputs. The DPCF constructs cannot help identify undefined inputs.

The effect of the hardware assumption we made is now clear. NPCF can detect undefined inputs, so it essentially has negation with respect to bottom (*cf.* Figure 14(b)). Therefore, NPCF resembles De Morgan circuits.

Since DPCF and NPCF operate on a different domain than boolean circuits, we need a way to take a function computed by a boolean circuit, say $f : \{0,1\}^n \rightarrow \{0,1\}$ and view it as a function on PCF domain elements, say $\perp, tt$ (the choice of tt rather than ff is arbitrary). Let us call the equivalent of f in the PCF world f_{pcf}. Conversely, given $g : \{\perp, tt\}^n \rightarrow \{\perp, tt\}$ in PCF, we define g_{bool} to be its counterpart in the circuit world.

DPCF and NPCF are at least as powerful as boolean circuits. Given a circuit, we can divide it into "blocks," each implementable by a DPCF (NPCF) function, and select a suitable application order that does not duplicate work, thus constructing an equivalent program. For the reasons listed above, monotone

circuits are as powerful as DPCF when presented with undefined inputs.

Proposition 19 *Given a monotone (De Morgan) circuit computing f we can construct from it a DPCF (NPCF) program computing f_{pcf}, with the same dimensions in the circuit semantics.*

Proof. We only discuss the monotone case here as the other case is similar.

It is clear that the only problems we might encounter occur when the fan-out is larger than 1 for some nodes. If the fan-out is always 1, the circuit is a tree, and can be represented by a formula. If the fan-out is larger than 1 for some node, then we have to be careful we do not duplicate work in the DPCF program. Since a link in the circuit is equivalent to an application in DPCF, we want to figure out an application order such that in the circuit semantics we end up with the same structure as the original circuit.

We give an algorithm to construct the DPCF program. First, we identify all nodes with fan-out > 1 in the circuit. Then, beginning at the bottom, we identify all the parts of the circuit which are trees. That is, we go as far as possible up the circuit in all directions, until we get to the recipient of a multiple output. This determines one DPCF function. We continue in similar fashion from the origins of the multiple outputs. We have now divided the circuit into blocks, and all we need to do is figure out an application order.

We start with the last block in the circuit (the one containing the output node); call it F_0. Let N be the set of all of its immediate neighbors. F_0 will be a function of $|N|$ arguments. We apply F_0 to all elements of N from which there is no path going into another element of N. This works because there are no cycles in the circuit. We continue like this with all elements from N which were used, and which receive inputs. A minor issue is the ordering of the inputs when doing the applications. We need to order them so that the element of N for which there exists the longest path into F_0 is the last input. In case of ties the order is not important.

We also have to make sure the variable names are the same in different blocks when they have to be bound to the same input. For instance, consider the following case: we have three nodes, F_0, F_1, and F_2, with F_1 having a link into F_0, and F_2 having links into both F_0 and F_1. Then the nodes will have the form $F_0 \equiv \lambda xy.\ f_0$, $F_1 \equiv \lambda z.\ f_1$, and the application order will be $F_0F_1F_2$. When we "hook up" F_0 and F_1 we have to change all the z's in F_1 to y's. Equivalently, we could have $F_1 \equiv \lambda y.\ f_1$ and use a β substitution rule that allows variable capture.

This procedure yields a DPCF program whose circuit semantics quite obviously looks the same as the original circuit, and therefore has the same size

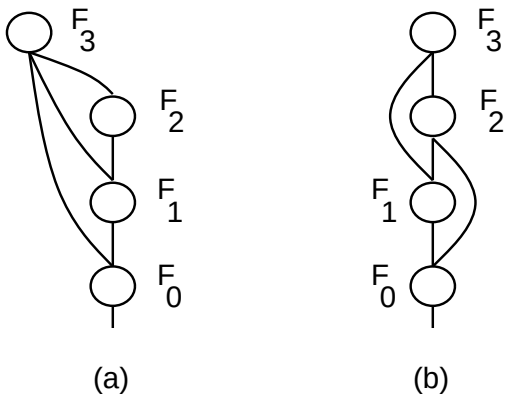

Fig. 16. Two circuits

and depth. □

Example 20 We give an example to illustrate the algorithm described above. Figure 16 shows the block structure of two circuits. Case (a) would give rise to the following program P_a (for simplicity, we assume β substitution with variable capture, as discussed above):

$$F_0 \equiv \lambda xy.\ f_0, \quad F_1 \equiv \lambda zx.\ f_1, \quad F_2 \equiv \lambda x.\ f_2$$
$$P_a \equiv F_0(F_1F_2)F_3.$$

Case (b) would result in the construction of the following P_b:

$$F_0 \equiv \lambda xy.\ f_0, \quad F_1 \equiv \lambda yz.\ f_1, \quad F_2 \equiv \lambda y.\ f_2$$
$$P_b \equiv F_0F_1F_2F_3.$$

□

Now we show that monotone circuits are as powerful as DPCF for a certain class of functions. For our applications, we do not need the equivalent statement for De Morgan circuits and NPCF programs.

Proposition 21 *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.*

Proof. Let us assume the function f is not constant, otherwise the proof is trivial. We can divide the constructs of DPCF into sequential and parallel. There is only one parallel construct, query. The sequential constructs, if presented with an undefined input, will all return $\bot$. Therefore, in order to obtain any result when computing f we must use query.

We can translate the tt in the input to anything else we would like: ff, any integer. We cannot translate $\bot$. The functions computable in DPCF on $\bot$ and any other non-bottom input are the same: AND, OR, constant functions. Since we must use query to make any progress, and since queries on $\bot$ and

any other input are all the same, it doesn't matter what the other input is. So translating the tt from the input makes no difference. We would simply have to translate back at the output.

We can then eliminate the unnecessary translations and leave only the queries. The circuit semantics of the resulting program with $\perp$, tt inputs changed to 0,1 inputs respectively, is a valid monotone circuit computing f_{bool}. $\Box$

Note that we can consider the DPCF program for f to be conditional-free, therefore, according to Propositions 15,16, the result implies that a boolean circuit can compute f_{bool} with at least the same efficiency that DPCF can compute f under either parallel c-b-v or c-b-s.

9 Applications

Given the correspondence we have established between DPCF, NPCF, and boolean circuits, we can use strong results from complexity theory to prove equivalent statements about our programming languages. These results are applicable for either parallel c-b-v or c-b-s (*cf.* discussion at the end of previous section), and also contain implications about evaluation on the PRAM.

Improving on an earlier superpolynomial bound by Razborov, Tardos [33] proved a very strong separation result between monotone and De Morgan circuit complexity:

Theorem 22 *[Tardos] There exists a polynomial time computable monotone function whose monotone complexity is exponential.*

The monotone function discussed is the perfect matching function [10], which takes an adjacency matrix as input and returns 1 if the graph has a perfect matching. DPCF and NPCF programs would compute the same function on $\{\perp, tt\}^n$ inputs.

Since perfect matching is in P, there exists a De Morgan circuit of polynomial size for it [4]. By Proposition 19 there is an NPCF program for it which does a polynomial amount of work. Since the monotone complexity of perfect matching is exponential, by Proposition 21 any DPCF program for it will do an exponential amount of work. Therefore, we have:

Proposition 23 *There exists a function computable by an NPCF program with polynomial work, but for which the best DPCF program does exponential work.*

By Propositions 17,18, the result can be stated in term of execution time on the PRAM: there exists a function computable by both DPCF and NPCF but which requires exponentially more time to execute on a PRAM for DPCF than NPCF.

A strong separation result between monotone and De Morgan circuits is also known for circuit depth [27].

Theorem 24 *[Raz & Wigderson] There is a monotone function in NC^1 that has no monotone NC circuits.*

The function referred to in Theorem 24 is a variant of the matching function (matching of size $n/3$, where n is the number of vertices in the graph). By a similar argument as above we can apply this result to NPCF and DPCF programs.

Proposition 25 *There exists a function computable by an NPCF program in logarithmic time, and for which the best DPCF program takes more than polylogarithmic time.*

10 Conclusions

We have defined a new, *intensional* denotational semantics for functional languages, circuit semantics. Circuit semantics associates a gate with each basic construct of the language, and takes the meaning of a program to be a circuit. The dimensions of the circuit enable reasoning about running time and work required for execution. We have established circuit semantics as a parallel complexity model, by comparing it to the time and work required to execute programs under several parallel evaluation strategies, and in the PRAM model. We have also used circuit semantics to obtain relative intensional expressiveness results for parallel extensions of PCF.

We have shown that deterministic query is intensionally more expressive than pif_t, which, in turn, is intensionally more expressive than *por* and pif_o. Thus, we have the beginnings of a hierarchy of intensional expressiveness for deterministic parallelism. In the process, we have exhibited languages which are extensionally but not intensionally equivalent. The constructs *por*, pif_o, and pif_t are interdefinable in the continuous function model of PCF. However, PCF + pif_t is intensionally more expressive than PCF + *por* (or pif_o). A natural question raised by this is whether there exists a language that is extensionally more expressive but intensionally less expressive (on the common subset of computable functions) than another language. The case of the Girard-Reynolds system F versus Gödel's system T might be an example of

this, but the matter is not settled yet (*cf.* [9]).

In order to compare deterministic and nondeterministic query, we were forced to make an assumption about having the ability to detect undefined inputs. Though somewhat unaesthetic, this assumption allowed us to view the question as a similar one from complexity theory, that of comparing monotone and De Morgan boolean circuits. After establishing a connection between the dimensions of a program under the circuit semantics and the complexity and depth of a corresponding boolean function, we were able to show that nondeterministic query is intensionally more expressive than its deterministic counterpart: it can lead to exponentially faster programs, and also programs that do exponentially less work.

Although we have used results from circuit complexity, have not "given anything back." It would be interesting to find out if the connection between DPCF programs and monotone circuits has some wider applicability in the area of circuit complexity. It seems unrealistic to hope that it would be easier to prove, say, strong lower bounds for the complexity of slice functions [34] by examining DPCF programs with undefined inputs. But perhaps the more general connection between functional parallel programs with undefined inputs and boolean circuits can be fruitfully exploited.

References

[1] M. Ajtai, J. Komlós, E. Szemerédi, An $O(n \lg n)$ sorting network, in: *Proc. ACM Symposium on the Theory of Computation, 1983*, 1–9.

[2] G. Berry and P.-L. Curien, Sequential algorithms on concrete data structures, *Theoretical Computer Science* 20 (1985) 265–321.

[3] G.E. Blelloch and J. Greiner, A parallel complexity model for functional languages, in: *Proc. ACM Conf. on Functional Programming Languages and Computer Architecture, 1995*.

[4] R.B. Boppana and M. Sipser, The complexity of finite functions, in: J. van Leeuwen ed., *Handbook of Theoretical Computer Science, Vol. A* (Elsevier, 1990), 749–804.

[5] S. Brookes and D. Dancanet, Sequential algorithms, deterministic parallelism, and intensional expressiveness, in: *Proc. ACM Symposium on Principles of Programming Languages, 1995*, 13–24.

[6] S. Brookes and S. Geva, Towards a theory of parallel algorithms on concrete data structures, *Theoretical Computer Science* 101 (1992) 177–221.

[7] P. Clote, A sequential programming language for the parallel complexity class NC, Boston College, Technical report BCCS-88-07, 1988.

[8] L. Colson, About Primitive Recursive Algorithms, in: *Proc. International Colloquium on Automata, Languages, and Programming, 1989*, 194–206.

[9] L. Colson, *Représentation intentionelle d'algorithmes dans les systèmes fonctionelles: une étude de cas*, Thése de Doctorat, Université Paris VII (1991).

[10] T.H. Cormen, C.E. Leiserson, R.L. Rivest, *Introduction to Algorithms* (MIT Press, 1990).

[11] P.-L. Curien, *Categorical Combinators, Sequential Algorithms, and Functional Programming* (Birkhäuser, 1993).

[12] D. Dancanet and S. Brookes, Programming language expressiveness and circuit complexity, in: *Internat. Conf. on the Mathematical Foundations of Programming Semantics, 1996*.

[13] M. Felleisen, On the expressive power of programming languages, *Science of Computer Programming* 17 (1991) 35–75.

[14] M.J. Fischer, The consensus problem in unreliable distributed systems (a brief survey), in: M. Karpinski, ed., *Proc. Internat. Conf. on Foundations of Computation Theory* (Springer, 1983) 127–140.

[15] J.-Y. Girard, A. Scedrov, P.J. Scott, Bounded linear logic: a modular approach to polynomial-time computability, in: *Theoretical Computer Science* 97 (1992) 1–66.

[16] J. Greiner and G.E. Blelloch, A provably efficient fully speculative parallel implementation, in: *Proc. ACM Symposium on Principles of Programming Languages, 1996*.

[17] C.A.R. Hoare, *Communicating Sequential Processes* (Prentice-Hall, 1985).

[18] J.E. Hopcroft and J.D. Ullman, *Introduction to Automata Theory, Languages, and Computation* (Addison-Wesley, 1979).

[19] P. Hudak and S. Anderson, Pomset interpretations of parallel functional programs, in: *Proc. ACM Conf. on Functional Programming Languages and Computer Architecture, 1987*, 234–256.

[20] N.D. Jones, Constant time factors do matter, in: *Proc. ACM Symposium on the Theory of Computation, 1993*, 602–611.

[21] N.D. Jones, Computability and complexity from a programming perspective, in: *Proc. Mathematical Foundations of Programming Semantics, 1995.*

[22] D. Leivant and J.-Y. Marion, Lambda calculus characterizations of poly-time, in: *Typed Lambda Calculi and Applications, 1993*, 274–288.

[23] L. Lamport, Using time instead of timeout for fault-tolerant distributed systems, in: *ACM Trans. on Programming Languages and Systems* 6(2) 1984, 254–280.

[24] J. McCarthy, A basis for a mathematical theory of computation, in: Braffort and Hirschberg, eds., *Computer Programming and Formal Systems* (North-Holland, 1963), 33–70.

[25] P. Panangaden and V. Shanbhogue, On the expressive power of indeterminate network primitives, Cornell University, Technical report TR 87-891, 1987.

[26] G.D. Plotkin, LCF considered as a programming language, *Theoretical Computer Science* 5 (1977) 223–56.

[27] R. Raz and A. Wigderson, Monotone circuits for matching require linear depth, in: *ACM Symposium on the Theory of Computation, 1990*, 287–292.

[28] P. Roe, Calculating lenient programs' performance, in: S.L. Peyton-Jones, G. Hutton, C. Kehler Holst, eds., *Functional Programming, Glasgow 1990* (Springer, 1990) 227–236.

[29] M. Rosendahl, Automatic complexity analysis, in: *Proc. Functional Programming Languages and Computer Architecture, 1989*, 144–156.

[30] D. Sands, Time analysis, cost equivalence and program refinement, in: *Proc. Foundations of Software Technology and Theoretical Computer Science, New Delhi, 1991*, 25–39.

[31] A. Stoughton, Interdefinability of parallel operations in PCF, *Theoretical Computer Science* 79 (1991) 357-8.

[32] S. Sur and W. Böhm, Functional, I-structure, and M-structure implementations of NAS benchmark FT, in: *Parallel Architectures and Compilation Techniques, 1994*, 47–56.

[33] É. Tardos, The gap between monotone and non-monotone circuit complexity is exponential, *Combinatorica* 7 (4) (1987), 141–2.

[34] I. Wegener, *The Complexity of Boolean Functions* (Wiley, 1987).

[35] W. Zimmermann, Automatic worst case complexity analysis of parallel programs, Technical Report ICSI 90-066, International Computer Science Institute, 1990.