Programming Language Expressiveness and Circuit Complexity

Denis Dancanet Stephen Brookes

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
{dancanet,brookes}@cs.cmu.edu
Phone: (412) 268–3075, Fax: (412) 268–5576

Abstract

This paper is a continuation of the work begun in [5] on establishing relative *intensional expressiveness* results for programming languages. Language 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. The question we address is: Does nondeterministic parallelism add intensional expressiveness? We compare deterministic and nondeterministic extensions of PCF, a simple functional programming language. We develop further the circuit semantics from our earlier work, and establish a connection between parallel PCF programs and boolean circuits. Using results from circuit complexity, and assuming hardware which can detect undefined inputs, we show that nondeterministic parallelism is indeed intensionally more expressive. More precisely, we show that nondeterministic parallelism can lead to exponentially faster programs, and also programs that do exponentially less work.

1 Introduction

We conduct an investigation of the relative intensional expressiveness of a deterministic and a nondeterministic parallel functional language. The languages we compare arose naturally from earlier work. The core functional language is a fragment of PCF [15], and the parallel primitive is query [6], a simple construct which allows several expressions to be evaluated at the same time. In [5] we established various intensional expressiveness results for parallel extensions of PCF, one of which was deterministic query. This raised the question if nondeterministic query is intensionally more expressive than the deterministic version. We answer it in the (qualified) positive here.

In order to be able to compare the two versions of query, we make a hardware assumption which is equivalent to being able 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 [9].

The main idea behind our intensional separation results was suggested by our earlier use of circuits as a semantics [5]: PCF programs are 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.

We develop our circuit semantics formally as a parallel complexity model, and extend it to take into account the query construct. We compare 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 [11], and also to the time and number of processors required to execute it on the PRAM model.

1.1 Related work

Several authors [18, 17, 19, 23] have used *profiling semantics*, i.e., semantics augmented with time and work information, to perform automatic complexity analysis. Roe [17] considered a parallel *lenient* language, and Zimmerman [23] a data-parallel language. Hudak and Anderson [11] 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, 10] provided intensional models 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 [10] 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 [8], 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 [20], 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 work on extensional comparisons of merging primitives in dataflow networks [14]. 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 intensional comparisons of parallel constructs. Thus, to our knowledge, this paper presents the first intensional comparison of deterministic and nondeterministic parallel constructs.

1.2 Outline of the paper

Section 2 describes the programming languages we are comparing; Section 3 develops the circuit semantics we introduced in [5] and extends it to cover the query construct; Section 4 discusses how to compare our two languages; Section 5 establishes the connection between between boolean circuits and our programming languages; Section 6 applies the connection to obtain intensional expressiveness results.

2 PCF and parallel extensions

PCF [15] is a typed λ-calculus with two ground types, booleans (o) and integers (ι). The syntax of PCF terms is given by the grammar:

$$M ::= c \mid x^\sigma \mid \lambda x^\sigma.\, M \mid MM$$

We omit the conventional typing rules, because of space limitations. We expand the traditional set of constants to include arithmetic operations and integer comparison functions, but we do

$por \equiv \lambda xy.$ query (x, y) is
$(tt, _) \Rightarrow tt$
$\mid (_, tt) \Rightarrow tt$
$\mid (ff, ff) \Rightarrow ff$

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

query : $(\sigma_1 \times \cdots \times \sigma_n) \rightarrow \tau$
query $(x_1^{\sigma_1}, \ldots \sigma_n)$ is
$p_1 \Rightarrow M_1^\tau$
$\ldots$
$p_k \Rightarrow M_k^\tau$

Figure 1: (a) parallel-or, (b) $not_\perp$, (c) query syntax

not consider recursion. This omission simplifies the definition of a semantics for the extension of PCF with a nondeterministic construct. However, since we still need to talk about undefined (or "missing") inputs, we introduce "undefined" constants Ω^σ, one for each type σ. Our constants are:

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

The standard denotational semantics for PCF is given by $\mathcal{D}$: Terms $\rightarrow$ Environments $\rightarrow \bigcup D_\sigma$; D_σ is a family of domains including the flat domains of booleans (D_{bool}) and integers (D_{int}), and such that $D_{\tau_1 \rightarrow \tau_2} = [D_{\tau_1} \rightarrow D_{\tau_2}]$, the continuous function space (see [15] for details).

2.1 Query

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 1(a)). The construct is quite general and can be studied outside of the context of concrete data structures. In this paper, we consider extensions of PCF with query.

The general form of the query syntax is shown in Figure 1(c). 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. Patterns are 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 should 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 $\mathcal{D}[\![v_i]\!] \sqsupseteq \mathcal{D}[\![w_i]\!]$. We extend the standard semantics with $\mathcal{D}[\![_]\!]\rho = \perp$, 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 $\mathcal{D}[\![M_i]\!] = \mathcal{D}[\![M_j]\!]$. An example of a nondeterministic query is Figure 1(b).

2.2 Deterministic query

Our formal semantics provides for the parallel evaluation of the patterns. The determinism restriction makes it fairly easy to define a semantics, shown in Figure 2. We use Q to refer to the general

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

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

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

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

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

Figure 2: Denotational semantics for deterministic query

form of query, from the previous section. *Amb* is McCarthy's ambiguity operator [13]:

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

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. It keeps track of whether the pattern match succeeds and of any bindings generated in the process.

2.3 Nondeterministic query

The semantics from Figure 2 also makes sense for nondeterministic query, but then the *amb* operator can be presented with distinct defined inputs. However, under this interpretation we do not get any increase in intensional power, because a program must compute a function for the purpose of intensional comparisons.

Proposition 2.1 *Under the semantics of Figure 2, nondeterministic query is not intensionally more expressive than deterministic query.*

So we must find another interpretation. One possibility is shown in Figure 3. 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 2 except for a "don't care" symbol. Let us say that such a symbol is found a 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

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

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

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

Figure 3: Semantics for nondeterministic query

pattern p without location j), then the meaning of matching x^σ to "$_$" is a *poll* of the input. *Poll* is a nondeterministic construct [14] 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 that 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 take a detour into circuit semantics.

3 Circuit semantics

In [5] we introduced the notion of viewing a PCF program as a circuit. The basic idea 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 will be a circuit, which we can execute either top-down or bottom-up, and we can use its size and depth to reason about its complexity.

Figure 4(a)(b) shows the circuits representing constants and variables. The truth values and the integers are represented by nodes with no inputs. There could be several outputs (this is true for all other nodes; fan-out is unlimited). Each of the functional constants is a node with the required number of inputs. A variable is a wire, or for higher-order functions, a circuit labelled with the variable name. Figure 4(c) shows an input that is ignored; we need such a convention to write functions like the K combinator. The representation of conditionals 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 4(d)(e) shows the

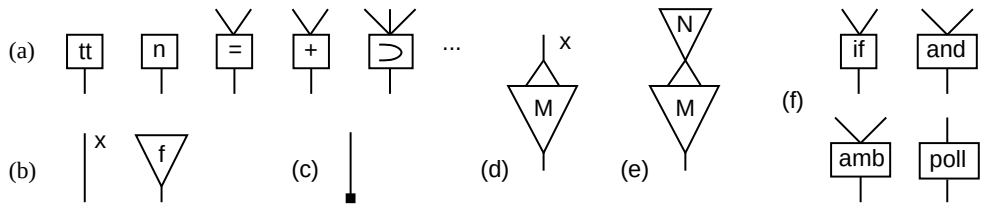

Figure 4: Circuits for (a) constants, (b) variables, (c) ignored inputs, (d) abstraction, (e) application, (f) query

representation of lambda abstraction and application. This points out the other major difference with dataflow networks: there are no application nodes. Note that our representation builds in sharing of arguments of ground type (functional arguments are not shared).

We define a model for the dimensions of a circuit. Figure 6 in the Appendix is an extended semantics for PCF, returning step-counting versions of a term, *S-Terms* (cf. *t-programs* in [18]), 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. We can prove by induction on terms that:

Proposition 3.1 *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.*

The circuit dimensions are most closely related to the work and time required to evaluate a program under call-by-speculation (c-b-s) [11, 17, 10]. Given $f\ x$, c-b-s evaluates f and x in parallel, but does not require that the evaluation of x be complete before the body of f is evaluated. Thus c-b-s allows fully pipelined parallelism. If x is 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 the *minimum* and *maximum* time to evaluate an expression.

Figure 7 in the Appendix shows a profiling semantics for c-b-s, in the style of [10]. The possible results of an evaluation are values, ranged over by v, which are either constants or function closures:

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

The judgments are of the form $\rho, d \vdash M \longrightarrow_{\text{cbs}} v; d', \hat{d}', w$, 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 of the depth at which a value becomes available. The constants chosen for the depth and work in several of the rules are different from [3], as we want to obtain an exact match with our circuit semantics. 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. We can prove by induction on terms that:

Proposition 3.2 *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 $d = t$.*

The circuit semantics of query is a straightforward implementation of the semantics presented in Section 2. Accordingly, we need to expand our set of basic gates with those of Figure 4(f). An

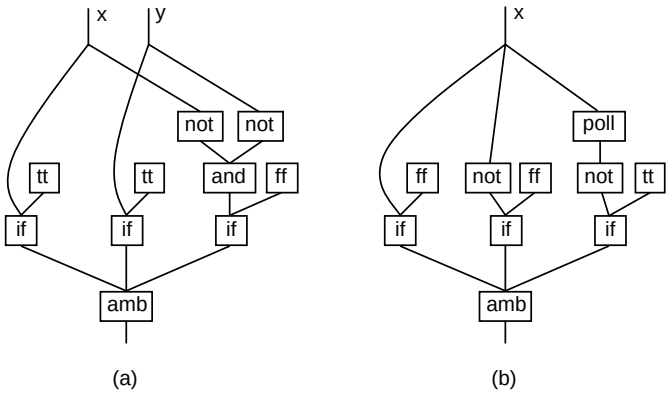

Figure 5: Circuit semantics for: (a) *por*, (b) $not_\perp$

"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 5(a) shows the circuit semantics of parallel-or. The *not* gate is shorthand for $\supset_o x$ *ff tt*. Figure 5(b) shows the circuit semantics of $not_\perp$ from Figure 1(b), which essentially performs negation with respect to $\perp$ (note how "_" is used to represent $\perp$).

4 On the method and the metric

The circuit semantics provides the metric for our comparisons. The circuit semantics of a program P defines its parallel complexity: the parallel time and the parallel work required to execute it. Accordingly, we can define two notions of intensional expressiveness: work-expressiveness and time-expressiveness. We are interested in asymptotic complexity: for a function f of n arguments, we want to compare the size and depth of circuits representing DPCF and NPCF programs which compute f as functions of n.

We want our results to apply to other parallel models, so we relate circuit semantics to the PRAM [7]. Suppose we have a program M whose parallel complexity according to the circuit semantics is size s and depth d. We can imagine the process of running M on a PRAM in two parts: construction and execution. First we construct the circuit semantics of M, then we execute it. We consider the simpler execution part first, which is similar to Brent's Theorem [7].

Proposition 4.1 *[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 $O(s/p + d)$.*

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 time to generate circuits for each of the two kinds of queries.

Proposition 4.2 *[Construction equivalence] It takes the same time on a p-processor PRAM to construct circuits for deterministic and nondeterministic interpretations of a query.*

Our notion of intensional expressiveness involves comparing the complexity of computing *functions*, so we have to focus on nondeterministic programs which return deterministic values. Since a purely nondeterministic interpretation of query fails to be intensionally more expressive (Proposition 2.1), we chose an interpretation that allows us to get deterministic answers, assuming we can detect undefined inputs. Such an assumption has been made in other areas, e.g., consensus problems in unreliable distributed systems [9]. Without being able to detect the failure of a process to send a message, e.g., in a fully asynchronous model, the consensus problem is not solvable. The physical realization of the assumption is quite reasonable, and does not even require fully synchronous hardware [12]. 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), our question has become: Does the ability to detect bottom in NPCF make it intensionally more work- or time-expressive than DPCF?

To answer this, we have to exhibit a function computable by *both* NPCF and DPCF, but which can be computed faster or with less work by NPCF. This is difficult because 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 [22].

Such considerations led us to examine more closely the similarity between DPCF and NPCF programs and boolean circuits.

5 A connection with 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 [22]. The De Morgan basis is complete.

The main idea of the connection is very simple: 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 domain. 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 5(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 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 5.1 *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.*

Proposition 5.2 *If DPCF can compute $f : \{\perp, tt\}^n \rightarrow tt$, then monotone circuits can compute f_{bool} with a circuit whose dimensions are the same as the circuit semantics of the DPCF program.*

6 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 time and work under c-b-s evaluation, and also contain implications about evaluation on the PRAM.

Improving on an earlier superpolynomial bound by Razborov, Tardos [21] proved a very strong separation result between monotone and De Morgan circuit complexity:

Theorem 6.1 *[Tardos] There exists a polynomial time computable monotone function whose monotone complexity is exponential.*

The monotone function discussed is the perfect matching function [7], 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 5.1 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 5.2 any DPCF program for it will do an exponential amount of work. Therefore, we have:

Proposition 6.2 *There exists a function computable by an NPCF program with polynomial work, but for which the best DPCF program does exponential work.*

By Propositions 4.1,4.2, 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 [16].

Theorem 6.3 *[Raz & Wigderson] There is a monotone function in NC^1 that has no monotone NC circuits.*

The complexity classes NC^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 fan-in; $\mathrm{NC} = \bigcup_{k \geq 0} \mathrm{NC}^k$ (see [4] for details). The function referred to in Theorem 6.3 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 6.4 *There exists a function computable by an NPCF program in logarithmic time, and for which the best DPCF program takes more than polylogarithmic time.*

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

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

[9] 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.

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

[11] 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.

[12] 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.

[13] 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.

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

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

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

[17] 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.

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

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

[20] 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.

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

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

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

A Extended semantics for PCF

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

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

depth: Terms $\to$ S-Terms
$\mathit{depth}(tt) = 1$
$\mathit{depth}(n) = 1$
$\mathit{depth}(M_1 = M_2) = 1 + \max(\mathit{depth}(M_1), \mathit{depth}(M_2))$
$\mathit{depth}(M_1 + M_2) = 1 + \max(\mathit{depth}(M_1), \mathit{depth}(M_2))$
$\mathit{depth}(\supset_\iota\ M_1\ M_2\ M_3) = 1 + \max(\mathit{depth}(M_1), \mathit{depth}(M_2), \mathit{depth}(M_3))$
$\mathit{depth}(x^\sigma) = x^\sigma$
$\mathit{depth}(M_1\ M_2) = \mathit{depth}(M_1)\ \mathit{depth}(M_2)$
$\mathit{depth}(\lambda x^\sigma.\ M) = \Lambda x^\sigma.\ \mathit{depth}(M)$

size: Terms $\to$ Environments $\to$ (S-Terms $\times$ Environments)
$\mathit{size}(tt)\eta = (1, \eta)$
$\mathit{size}(n)\eta = (1, \eta)$
$\mathit{size}(M_1 = M_2)\eta = (1 + s_1 + s_2, \eta'')$, where $(s_1, \eta') = \mathit{size}(M_1)\eta$, $(s_2, \eta'') = \mathit{size}(M_2)\eta'$
$\mathit{size}(M_1 + M_2)\eta = (1 + s_1 + s_2, \eta'')$, where $(s_1, \eta') = \mathit{size}(M_1)\eta$, $(s_2, \eta'') = \mathit{size}(M_2)\eta'$
$\mathit{size}(\supset_\iota\ M_1\ M_2\ M_3)\eta = (1 + s_1 + s_2 + s_3, \eta''')$, where
$(s_1, \eta') = \mathit{size}(M_1)\eta$, $(s_2, \eta'') = \mathit{size}(M_2)\eta'$, $(s_3, \eta''') = \mathit{size}(M_2)\eta''$

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

$\mathit{size}(M_1\ M_2)\eta = (s_1 s_2, \eta'')$, where $(s_1, \eta') = \mathit{size}(M_1)\eta$, $(s_2, \eta'') = \mathit{size}(M_2)\eta'$
$\mathit{size}(\lambda x^\sigma.\ M)\eta = (\Lambda x^\sigma.\ s_1, \eta')$, where $(s_1, \eta') = \mathit{size}(M)\eta$

Figure 6: Extended semantics for PCF

B Definition of call-by-speculation

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

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

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

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

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

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

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

Figure 7: Profiling semantics for call-by-speculation