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

$$
\begin{array}{lll}
\mathit{por} \equiv \lambda xy.\ \text{query } (x,y) \text{ is} & \mathit{not}_\perp \equiv \lambda x.\ \text{query } (x) \text{ is} & \text{query} : (\sigma_1 \times \cdots \times \sigma_n) \rightarrow \tau \\
\qquad\qquad (\mathit{tt},\ \_) \Rightarrow \mathit{tt} & \qquad\qquad \mathit{tt} \Rightarrow \mathit{ff} & \text{query } (x_1^{\sigma_1}, \ldots \sigma_n) \text{ is} \\
\qquad\qquad \mid (\_,\ \mathit{tt}) \Rightarrow \mathit{tt} & \qquad\qquad \mathit{ff} \Rightarrow \mathit{ff} & \qquad p_1 \Rightarrow M_1^\tau \\
\qquad\qquad \mid (\mathit{ff},\ \mathit{ff}) \Rightarrow \mathit{ff} & \qquad\qquad \_ \Rightarrow \mathit{tt} & \qquad \ldots \\
 & & \qquad p_k \Rightarrow M_k^\tau
\end{array}
$$

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:

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

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

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

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

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

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

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

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: $tt \ \wedge \ tt = tt$, $f\!f \ \wedge \ \bot = f\!f$, $\bot \ \wedge \ f\!f = f\!f$.

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

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

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

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

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:

$$\textit{poll}\ \bot = f\!f, \qquad \textit{poll}\ x = tt, \text{ if } x \neq \bot.$$

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