Sequential Algorithms, Deterministic Parallelism, and Intensional Expressiveness*

Stephen Brookes Denis Dancanet

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
{brookes,dancanet}@cs.cmu.edu

Abstract

We call language L_1 *intensionally more expressive* than L_2 if there are functions which can be computed faster in L_1 than in L_2. We study the intensional expressiveness of several languages: the Berry-Curien programming language of sequential algorithms, CDS0, a deterministic parallel extension to it, named CDSP, and various parallel extensions to the functional programming language PCF. The paper consists of two parts.

In the first part, we show that CDS0 can compute the minimum of two numbers n and p in unary representation in time $O(min(n,p))$. However, it cannot compute a "natural" version of this function. CDSP allows us to compute this function, as well as functions like parallel-or. This work can be seen as an extension of the work of Colson [7, 8] with primitive recursive algorithms to the setting of sequential algorithms.

In the second part, we show that deterministic parallelism adds intensional expressiveness, settling a "folk" conjecture from the literature in the negative. We show that CDSP is more expressive intensionally than CDS0. We also study three parallel extensions to PCF: parallel-or (*por*) and parallel conditionals on booleans (pif_o) and integers (pif_ι). The situation is more complicated there: pif_ι is more expressive than both pif_o and *por*. However, pif_ι still is not as expressive as the deterministic query construct of CDSP. Thus, we identify a hierarchy of intensional expressiveness for deterministic parallelism.

*This research was sponsored by the Office of Naval Research under Grant No. N00014-93-1-0750.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

To appear in the Proceedings of the 22nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, January 1995

1 Introduction

We are interested in establishing relative *intensional* expressiveness results for programming languages. Most work in the past has focussed on *extensional* expressiveness: Language L_1 is *extensionally more expressive* than L_2 if there are functions that are computable in L_1 but not computable in L_2. We say that L_1 is *intensionally more expressive* than L_2 if there are functions computable *faster* in L_1 than in L_2. 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 [13]; and there are certain problems for which there exist faster CRCW PRAM algorithms than EREW PRAM algorithms [10]. Our work compares programming languages, not their underlying computation models.

In the first part of this paper, we study the expressibility of the minimum function, which computes the minimum of two natural numbers represented in unary form (0, $S(0)$, ..., where S stands for successor). We look at various algorithms computing minimum, which agree when the inputs are fully defined (as they must, since they all compute minimum), but may disagree on undefined or partial inputs.

A natural way to define minimum is by the following rewrite system:

$$min(x,0) = 0$$
$$min(0,x) = 0$$
$$min(S(x),S(y)) = S(min(x,y))$$

We need to distinguish between the function min (the least function satisfying the rewrite rules above), and an algorithm for min, which we denote min_a. Intuitively, the algorithm based on the above rewrite rules computes its result in time $O(min(n,p))$ (it takes exactly $min(n,p)+1$ steps). One can formalize this by

giving an operational semantics, and defining a notion of cost (*cf.* Colson [8]).

The questions we are interested in are: Is it possible to write a program to compute the minimum of n, p in time $O(min(n,p))$ in Berry and Curien's language of sequential algorithms, CDS0? Is this possible in a language of parallel algorithms, CDSP, obtained by generalizing the valof construct of CDS0 to a parallel form of query? In the second part of the paper we consider the following more general questions: Does the parallel query construct of CDSP give added intensional expressiveness over CDS0? How does it relate to the expressiveness of parallel extensions to PCF?

The rest of the paper is organized as follows: First, we describe intensional semantics and the domain of lazy natural numbers. We give a brief overview of sequential algorithms, concrete data structures, and the programming languages CDS0 and CDSP. Then we review Colson's results concerning primitive recursive algorithms. We describe our results with CDS0, CDSP, and parallel PCF. We end with conclusions.

2 Background

2.1 Intensional Semantics

Traditionally, most denotational semantic models of programming languages have been extensional, designed to express only the input/output behavior of a program. We are interested in reasoning about intensional aspects (*e.g.,* complexity), so we need semantic models that contain more computational information. This can be achieved in many ways. We outline just a few possibilities:

- We could take the meaning of a program to be a function on a richer domain (*e.g.,* [4, 7]) whose structure permits us to deduce information about computation strategy.
- We could take the meaning to be a pair consisting of a function and an object conveying intensional information; this object could represent the cost of evaluating the function, or could be a function from inputs to costs (*e.g.,* [12, 19]).
- We could dispense with functions as meanings altogether, and use *algorithms* instead (*e.g.,* [2]).

An important point to note is that intensionality is relative. A model can be more intensional than another one, if its elements convey extra computational detail.

For a simple example consider the semantics of primitive recursive ($\mathcal{PR}$) algorithms. $\mathcal{PR}$ algorithms are just syntax for expressing $\mathcal{PR}$ functions [16]. The syntax is in the form of a rewrite system (see Colson [7, 8] for a formal definition). Consider the following two algorithms for integer addition in unary representation [8]:

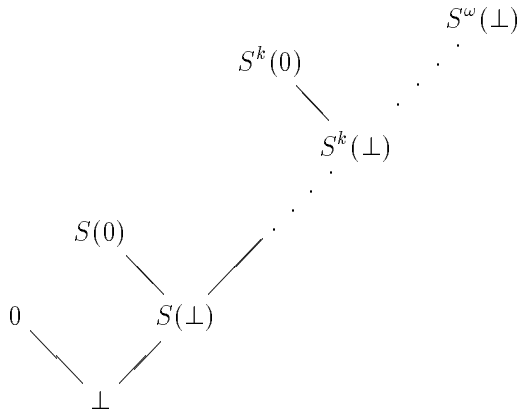

Figure 1: The lazy natural numbers

$$add1(0,y) = y$$
$$add1(S(x),y) = S(add1(x,y))$$

$$add2(x,0) = x$$
$$add2(x,S(y)) = S(add2(x,y))$$

The standard extensional denotational semantics for $add1$, $add2$ maps them both into the addition function of type $N^2 \to N$, where N is the flat domain of natural numbers. A simple intensional semantics may be provided by using the lazy natural numbers [7, 8, 9]. The domain $LNAT$ is shown in Figure 1. $LNAT$ captures the temporal aspect of finding out what an input is. At $S^k(\bot)$ we don't know yet if we have the number $S^k(0)$, or something larger (at least $S^{k+1}(\bot)$). This intensional semantics is sufficient to distinguish between the two addition algorithms. Using the meaning function $[\![\ ]\!]$ from [8, 9] (which makes the meaning $\bot$ when an algorithm tries to recur on $\bot$) we have:

$$[\![add1]\!](S^2(\bot),S(\bot)) = S^2(\bot)$$
$$[\![add2]\!](S^2(\bot),S(\bot)) = S(\bot)$$

The $LNAT$ semantics is richer than the N semantics, and contains intensional information; the above equations can be interpreted as showing that at some point, $add2$ tries to evaluate part of its second argument before the first, whereas $add1$ looks at its first input first. Although the $LNAT$ semantics still represents the meanings of $add1$ and $add2$ as functions (from $LNAT \times LNAT$ to $LNAT$), it conveys implicit information about computation strategy. In the next section we describe Berry-Curien sequential algorithms, which can be used to provide a more explicitly intensional model for $\mathcal{PR}$ algorithms.

2.2 Sequential algorithms and concrete data structures

Sequential algorithms on concrete data structures provide an intensional semantics for sequential programming languages. In contrast with the traditional extensional semantics for such languages, continuous functions are replaced by sequential algorithms, and Scott domains by concrete data structures. We provide a very brief overview, along the lines of Berry and Curien's work. The interested reader is referred to [2, 11] for details and definitions. An alternative description in terms of decision trees can be found in [14].

2.2.1 Concrete data structures

Concrete data structures and their domain-theoretic counterparts (concrete domains) were developed by Kahn and Plotkin [15] in order to distinguish between function domains and domains of the data on which they compute. They provide an abstract framework for modelling incremental sequential computation.

A concrete data structure (cds) consists of a set of named *cells*, which can hold *values*, and an *accessibility* relation governing the order in which the cells can be *filled* with values. A cell c filled with a value v is called an *event*, written $c = v$. A set of events satisfying certain conditions (no cell is filled more than once, accessibility relation is respected) is called a *state*. The set of states of a cds M ordered by set inclusion form a concrete domain $\langle D(M), \subseteq \rangle$.

Example 2.1 We define $BOOL$, the cds of booleans. There is one cell called B, which can be filled with either tt or ff. The set of states of this cds is:

$$\{\{\}, \{B = tt\}, \{B = ff\}\}.$$

Note that $\langle D(BOOL), \subseteq \rangle$ is isomorphic to the flat domain of booleans. □

Example 2.2 We define $LNAT$, the cds of lazy natural numbers. It has cells b_n, for $n \geq 0$, values 0 and 1, and the following accessibility relation: b_0 is initial (no precondition), and $\{b_i = 1\} \vdash b_{i+1}$ (filling a cell with 1 enables the next cell). Intuitively, filling a cell with 1 means there might be more to follow, whereas 0 means we're done. $\langle D(LNAT), \subseteq \rangle$ is isomorphic to the domain $LNAT$ from the previous section. The encoding of the lazy natural numbers is:

$$\begin{aligned} S^n(\bot) &= \{b_i = 1 \mid i < n\}, \\ S^n(0) &= \{b_i = 1 \mid i < n\} \cup \{b_n = 0\}, \text{ for } n \geq 0, \\ S^\omega(\bot) &= \{b_i = 1 \mid i \geq 0\}. \end{aligned}$$

□

Using the cds framework, Kahn and Plotkin defined a notion of sequential function. A continuous function f from $D(M)$ to $D(M')$ is *sequential at* x if for each cell c' accessible in $f(x)$ either (i) no cell is accessible in x, or (ii) there is a cell c accessible at x that must be filled in any state y that is a superset of x such that c' is filled in $f(y)$. The cell c is called a *sequentiality index* of f at x for c'. A function is *sequential* if it is continuous and sequential at every x in its domain. Intuitively, this definition captures the notion that a sequential function is at any point dependent on one of its inputs; if that input diverges, the function will diverge.

2.2.2 Sequential algorithms

Berry and Curien noted that Kahn-Plotkin sequential functions and concrete data structures fail to form a cartesian closed category. They introduced *sequential algorithms* on concrete data structures, and showed that the category of sequential algorithms and cds is cartesian closed.

Sequential algorithms can be viewed two ways: abstractly and concretely. Abstractly, a sequential algorithm is a pair consisting of a sequential function and a (sequential) computation strategy. If there are several ways of proceeding during the computation, the computation strategy picks out a particular one. Concretely, a sequential algorithm is a state of a cds of arrow type (the *exponentiation* cds).

Given two cds, M and M', the exponentiation cds $M \Rightarrow M'$ is defined[1] as follows: the cells are of the form xc', where x is a state of M and c' is a cell of M'; the values are of the form $valof\ c$ where c is a cell of M, or $output\ v'$ if v' is a value of M'. A state of $M \Rightarrow M'$ is a sequential algorithm.

Example 2.3 The state of $BOOL \Rightarrow BOOL$ that corresponds to the boolean negation is:

$$\begin{aligned} \{\{\}B &= valof\ B, \\ \{B = tt\}B &= output\ ff, \\ \{B = ff\}B &= output\ tt\}. \end{aligned}$$

The way to read this definition is: Given no information about the input and having to fill the output cell B, we ask what value the input cell B holds. If the input is true we output false and conversely. □

2.3 CDS0

The programming language CDS0 [1, 3, 11] is a direct implementation of the intensional denotational seman-

[1] This is only part of the definition; it omits reference to the accessibility conditions. The full definition can be found in [11], for instance.

tics presented above; hence, it is an intensional programming language of sequential algorithms. The name stands for Concrete Data Structures.

CDS0 is a lazy, polymorphic, higher-order, functional language with some original features:

- Uniformity of types. Everything in CDS0 is a state of a cds. This can be a state-constant or a higher-order algorithm. The algorithm syntax is just syntactic sugar for the state of a cds. Consequently, an algorithm can be evaluated without being applied to any argument. Operationally speaking, terms of non-ground type can be observed.
- Full abstraction. The denotational semantics of CDS0, which maps an algorithm to a state of the cds corresponding to its type (hence a CDS0 object) is fully abstract with respect to two different operational semantics (CDS01 and CDS02) [11].
- Semantics manipulation. Since the semantics of an algorithm is itself a CDS0 state it is possible to write algorithms which manipulate the *semantics* of other algorithms.

Example 2.4 As a simple example[2], we implement the cds of booleans and boolean negation in CDS0. The boolean negation algorithm is given in two forms: sugared (NOT) and un-sugared (NOT_STATE):

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

let NOT : BOOL -> BOOL = algo
      request B do
        valof B is
          tt : output ff
          ff : output tt
        end
      end
end;

let NOT_STATE : BOOL -> BOOL =
  {{}B    = valof B,
  {B=tt}B = output ff,
  {B=ff}B = output tt};
```

The *request* construct specifies which output cell we are computing, *valof* requests a value from an input cell, and *output* fills a cell with a value. □

The user can combine algorithms and states into expressions using the *categorical* combinators: application, composition, curry, uncurry, fix, pair, and product. Computation is lazy, demand-driven: the user types in an expression and enters a *request loop*. At this point the user may type in an output cell name and if the cell is filled in that expression its value will come back as a result. The lazy evaluation model enables us to compute with infinite structures.

Example 2.5 As a more advanced example, we implement the lazy natural numbers, and the successor algorithm, which are needed in what follows.

```
letrec LNAT = dcds
         cell B values 0,1
         graft (LNAT.s) access B = 1
end;
```

$LNAT$ is defined using recursion and *grafting*: a copy of $LNAT$ is included, tagging all cells with the specified tag. The first three cells and their access conditions are as follows:

```
B values 0, 1
(B.s) values 0, 1 access B=1
((B.s).s) values 0, 1 access (B.s)=1
```

Now let us define a few constants: $\perp$, 0, $S(\perp)$, and $S^\omega(\perp)$:

```
let Bot  : LNAT = {};
let Zero : LNAT = {B=0};
let S_bot : LNAT = {B=1};

let Srec : LNAT -> LNAT = algo
      request B do
        output 1
      end
      request ((B.$V).s) do
        valof (B.$V) is
          1 : output 1
        end
      end
end;

let S_omega_bot : LNAT = fix(Srec);
```

$S^\omega(\perp)$ is defined as the least fixpoint of the algorithm which in the base case fills B with 1, and recursively, if the previous cell contains 1, puts 1 into the current cell. The "name variable" `$V` matches any tag. The ability to use such variables for cell names and values is the source of polymorphism in CDS0.

Now we can write the successor algorithm. Its structure is just slightly more complicated than the algorithm for $S^\omega(\perp)$, but warrants further explanation because it is higher-order. Successor is defined as the fixpoint of a higher-order algorithm and it works as follows: If asked what B is, it immediately outputs 1 (the successor of anything is at least $S(\perp)$). In the general case, if asked what value an output cell holds, it asks what value the input cell immediately preceding it holds, and outputs the same value.

[2]The syntax for the CDS0 examples presented in this section is from our own CDS0 interpreter and is very similar to that in [3, 11].

```
let succ_rec :(LNAT -> LNAT) -> LNAT -> LNAT =
    algo
      request {}B do
        output output 1
      end
      request {}((B.$V).s) do
        output valof (B.$V)
      end
      request {(B.$V)=0}((B.$V).s) do
        output output 0
      end
      request {(B.$V)=1}((B.$V).s) do
        output output 1
      end
 end;

 let S : LNAT -> LNAT = fix(succ_rec);
```

□

2.4 CDS: A higher-level notation

As can be seen from the examples in the previous section, the syntax of CDS0 is very low-level. In fact, CDS0 was designed to serve as a "machine-language" for a syntactically ML-like language called CDS [1]. CDS was never fully described or implemented. For ease of presentation we assume an SML-like notation [17] for it, glossing over exactly how the translation to CDS0 might be accomplished (for a discussion see [3]). An important difference between CDS and SML is that in CDS pattern matching is not allowed on tuples, so that the sequential nature of the computation is more readily apparent.

2.5 CDSP: A parallel version of CDS

Brookes and Geva [5] extended Berry and Curien's work to the setting of *deterministic parallel algorithms* on cds. They generalized the valof construct of CDS0 which tests the value of one cell to a deterministic parallel *query* construct, which, intuitively, spawns off a number of valofs. More precisely, a query starts a number of parallel subcomputations and specifies conditions based on the results of the subcomputations under which the main computation may resume. We call the extension of CDS0 with the query construct CDSP (for CDS Parallel). A CDSP algorithm may be viewed as a *continuous* function paired with a (parallel) computation strategy.

Example 2.6 Query enables us to compute new functions. One example is parallel-or, which returns true if either of its arguments is true. Parallel-or, as the name implies, is not a sequential function. Here is how it would be implemented in CDSP, assuming the higher-level syntax:

algo por (b1, b2) = **query** (b1, b2) **is**
$$(tt, \_) \Rightarrow tt$$
$$| \ (\_, tt) \Rightarrow tt$$
$$| \ (ff, ff) \Rightarrow ff$$

□

In order to ensure determinism, all *consistent* (simultaneously satisfiable) branches of a query must have the same result; for example, the first two branches of the above algorithm result in the same output. This requirement is built into the syntax used in [5]. We use a simpler notation in order to avoid the extra complexity.

3 Colson's Results

Colson studied the expressibility of the minimum function in the context of primitive recursive ($\mathcal{PR}$) algorithms [7, 8]. He established that $\mathcal{PR}$ algorithms are inherently sequential: like sequential algorithms, they possess sequentiality indices. Moreover, $\mathcal{PR}$ algorithms are sequential in an even stronger sense. They suffer from "ultimate obstination" [8, 9]: at some point one argument must be chosen to be evaluated until the end. Using primarily the intensional denotational semantics based on $LNAT$, Colson proved two main results:

Proposition 3.1 *There is no $\mathcal{PR}$ algorithm a of arity 2 satisfying:*

$$[\![a]\!](S^n(\bot), S^p(\bot)) = S^{min(n,p)}(\bot).$$

Proposition 3.2 *There is no $\mathcal{PR}$ algorithm computing the minimum of two numbers n and p in unary representation, with time complexity $O(min(n,p))$.*

However, there are many $\mathcal{PR}$ algorithms which compute the minimum of two integers in unary representation. We define one below, using some auxiliary functions (see [16]):

$$pred(0) = 0$$
$$pred(S(x)) = x$$

$$sub(x, 0) = x$$
$$sub(x, S(y)) = pred(sub(x, y))$$

$$MIN(x, y) = sub(x, sub(x, y)).$$

Again we need to distinguish between the function MIN and an algorithm MIN_a for MIN. Note that in an operational interpretation of this definition, the algorithm $MIN_a(n,p)$ has a worst-case running time of $O(max(n,p))$. The function MIN agrees with min from Section 1 on the totally defined elements of the

lazy naturals. They differ on the partial elements, since in the $LNAT$ semantics we have:

$$\begin{aligned} min(S^n(\bot), S^p(\bot)) &= S^{min(n,p)}(\bot), \\ MIN(S^n(\bot), S^p(\bot)) &= \bot. \end{aligned}$$

We can view Proposition 3.1 as an extensional expressiveness result: $\mathcal{PR}$ algorithms can compute MIN but not min. Note that there are many other functions between min and MIN in the pointwise order. But it is the intensional aspect of Proposition 3.2 that is particularly interesting here: $\mathcal{PR}$ algorithms cannot compute minimum efficiently.

If we augment $\mathcal{PR}$ algorithms with functional arguments, we arrive at Gödel's system T. System T can not only compute new functions (*e.g.,* the Ackermann function), but can also compute minimum efficiently. Thus system T is more powerful than $\mathcal{PR}$ both extensionally and intensionally (*cf.* [8]).

Colson's results are the first intensional expressiveness results for programming languages of which we are aware.

4 Sequential, Parallel Algorithms, and Minimum

We expected to obtain results similar to Colson's in our study of sequential algorithms. After all, CDS0 is a sequential programming language by design: sequential algorithms compute sequential functions. It turns out, however, that sequential algorithms are sufficiently more powerful than $\mathcal{PR}$ algorithms to be able to compute minimum efficiently, but not powerful enough to compute the "natural" min function from the introduction. The parallel query construct of CDSP allows us to compute that function.

4.1 CDS0

We begin by showing that sequential algorithms cannot compute min. The proof follows standard lines (*cf.* [2, 5]).

Proposition 4.1 *There is no sequential algorithm computing min.*

Proof: A sequential algorithm computes a sequential function. But min is not sequential, since it has no sequentiality index at $(\bot, \bot)$ for output cell b_0. In other words, there is no input cell which must be filled in order for min to fill b_0. (Actually, min has no sequentiality index at any $(S^n(\bot), S^n(\bot))$ for b_n, $n \geq 0$.) Therefore, no CDS0 algorithm can compute min. □

But this does not mean we cannot compute minimum efficiently in CDS0. Recall that the problem with $\mathcal{PR}$ algorithms was that they become "fixated" on one input. Sequential algorithms allow us to keep alternating between the two inputs, examining one cell at a time.

We reason informally about the running time of sequential algorithms. It is possible to formalize these arguments by appealing to the operational semantics.

Proposition 4.2 *There is a sequential algorithm which computes the minimum of two numbers n and p in unary representation, and is of time complexity $O(min(n,p))$.*

Proof: The algorithm looks like a simple sequential version of the min function definition from the introduction. We choose the left input to evaluate first.

```
algo left_min (n1, n2) =
     case n1 of
          0 ⇒ 0
        | S(x) ⇒ case n2 of
                      0 ⇒ 0
                    | S(y) ⇒ S(left_min(x, y))
```

The algorithm has the following property:

$$[\![\text{left\_min}]\!](S^n(0), S^p(0)) = S^{min(n,p)}(0),$$

so it does compute the minimum, and it works in time $O(min(n,p))$ by alternating between the inputs and examining one cell at a time. □

Note that the algorithm also satisfies:

$$[\![\text{left\_min}]\!](S^n(\bot), S^p(\bot)) = S^{min(n,p)}(\bot),$$

so Colson's Proposition 3.1 fails as well in the context of sequential algorithms.

The key difference between left_min and min_a is illustrated by their behavior on pairs of a totally defined and a partial element, such as $(S^n(0), S^n(\bot))$ (they agree on all other inputs):

$$\begin{aligned} [\![\text{left\_min}]\!](S^n(0), S^n(\bot)) &= S^n(\bot) \\ [\![min_a]\!](S^n(0), S^n(\bot)) &= S^n(0) \end{aligned}$$

$$\begin{aligned} [\![\text{left\_min}]\!](S^n(\bot), S^n(0)) &= S^n(\bot) \\ [\![min_a]\!](S^n(\bot), S^n(0)) &= S^n(0) \end{aligned}$$

This comparison makes it clear that min is a parallel function: it must evaluate its inputs in parallel in order to be able to determine when either one is defined. Also note that [![left_min]!] fits between min and MIN in the pointwise order.

4.2 CDSP

The addition of the parallel query construct enables us to compute min, which is essentially a generalization of parallel-or to integer arguments. The program looks almost the same as the definition of the min function from the introduction:

algo min (n1, n2) =
  query (n1, n2) **is**
    (0, _) ⇒ 0
  | (_, 0) ⇒ 0
  | (S(x), S(y)) ⇒ S(min(x, y))

We then obtain the following, using the semantics of CDSP:

Proposition 4.3 *There is a CDSP program computing min.*

5 Deterministic Parallelism and Intensional Expressiveness

There appears to be a folk conjecture that deterministic parallelism is not "useful." The claim (see [6], for instance) is that even though deterministic parallel features may increase the extensional expressiveness of a language, they are expensive to use and the additional expressiveness is not useful in practice, because "it applies only to computations that are unbounded." In our terms, the claim is that deterministic parallelism may increase extensional, but not intensional expressiveness.

As we'll see in what follows, this conjecture is false. Deterministic parallelism does add intensional expressiveness. The deterministic query construct of CDSP is sufficiently general to allow a speedup in the computation of many different functions.

From our study of CDS0 and CDSP we are naturally drawn to a study of the sequential functional language PCF and its parallel extensions. The reason is the close connection between CDS0 and PCF: CDS0 is an intensional semantics for PCF [3]. In fact, CDS0 is extensionally more expressive than PCF, being able to express semantic-manipulation algorithms. Thus, the results we obtain with CDS0 and parallel extensions are likely to be mirrored with PCF and similar parallel extensions.

The situation is more complicated in the case of PCF: the different parallel extensions in the literature, although extensionally equivalent, are not intensionally equivalent. They do increase the computational power of PCF, but in different ways. They are also less powerful than CDSP's query.

In this second part of the paper we no longer work with integers in unary representation. We are interested in the running time of n-ary operations, such as adding n integers. Thus, our results should be more relevant to real programming languages.

5.1 CDSP

We give examples of two functions which can be computed faster in CDSP than in CDS0: n-ary disjunction and n-ary addition. They can be computed in logarithmic time, which is an improvement over the linear time achievable in CDS0. The two functions are very similar in structure. As with CDS0, we argue informally about the running time of CDSP programs.

The main idea is to construct a tree of processes. For notational simplicity, we define a separate function for each value of n, and we assume n is a (fixed) power of 2. We have already defined *por* for two arguments. Here is the general case:

algo $\mathrm{por}_n\ (b_1, \ldots, b_n) =$
  $\mathrm{por}\ (\mathrm{por}_{n/2}\ (b_1, \ldots, b_{n/2}),$
    $\mathrm{por}_{n/2}\ (b_{n/2+1}, \ldots, b_n))$

n-ary disjunction creates a tree of processes of depth $log\ n$. Addition for n arguments, $padd_n$, works similarly using addition on two arguments, *padd*, given by:

algo padd (x_1, x_2) = **query** (x_1, x_2) **is**
  $(v_1, v_2) \Rightarrow v_1 + v_2$

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.

When computing por_n or $padd_n$, in order to fill the output cell we query in parallel two cells. In order to fill those cells, we query two more for each. Intuitively, after a depth of $log\ n$ queries we reach our n inputs. Therefore, we compute the result in time $O(log\ n)$. In CDS0, since we must examine the inputs sequentially, we can only compute the result in time $O(n)$.

Proposition 5.1 *CDSP is intensionally more expressive than CDS0.*

Note that the previous proposition is unaffected by the fact that we are no longer using integers in unary representation. The function por_n would still be speeded up in CDSP even with unary representation for integers.

5.2 PCF and circuits

PCF [18] 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 is given by the grammar:

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

The constants traditionally included in the language are the following:

$$
\begin{array}{ll}
tt : o & \\
ff : o & \\
n : \iota & \text{(the integers, } n \geq 0\text{)} \\
isZero? : \iota \to o & \\
+1 : \iota \to \iota & \\
-1 : \iota \to \iota & \\
\supset_o : o \to o \to o \to o & \text{(boolean conditional)} \\
\supset_\iota : o \to \iota \to \iota \to \iota & \text{(integer conditional)} \\
Y_\sigma : (\sigma \to \sigma) \to \sigma & \text{(one for each } \sigma\text{)}
\end{array}
$$

For simplicity, we blur the distinction between numerals and integers, and use n to denote both.

The relevant rules of the operational semantics for the constants are:

$$
\begin{array}{l}
\supset_\sigma\ tt\ M_\sigma N_\sigma \to M_\sigma, \quad \text{for } \sigma = \iota, o \\
\supset_\sigma\ ff\ M_\sigma N_\sigma \to N_\sigma, \quad \text{for } \sigma = \iota, o \\
Y_\sigma M \to M(Y_\sigma M) \\
+1\ n \to n+1, \quad \text{for } n \geq 0 \\
-1\ (n+1) \to n, \quad \text{for } n \geq 0 \\
isZero?\ 0 \to tt \\
isZero?\ (n+1) \to ff, \quad \text{for } n \geq 0
\end{array}
$$

In addition to the standard constants listed above, 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.* [20]), but this would render some of the issues of interest to us moot, since we are not dealing with integers in unary representation.

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 corresponds to the depth of the circuit. We are only interested in programs of ground type so we need not worry about complications caused by higher-order programs. And third, circuits provide a visual and intuitive semantics.

The translation from PCF to circuits is simple. Figure 2 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 labelling an input with a function variable and using gates labelled with the function variable inside the circuit. There are

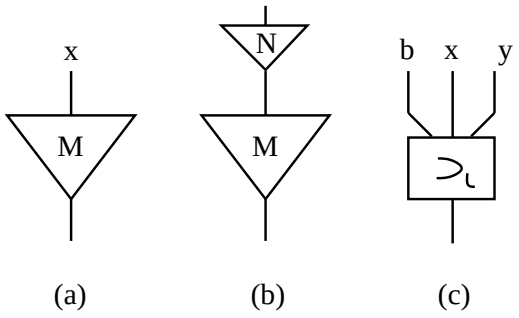

Figure 2: (a) $\lambda x.\ M$, (b) $(\lambda x.\ M)N$, (c) $\supset_\iota\ b\ x\ y$

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

Definition 5.3 *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 5.4 *The depth of a static circuit is equal to the height of the underlying tree.*

Definition 5.5 *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 5.6 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 program:

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

Figure 3 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 4.

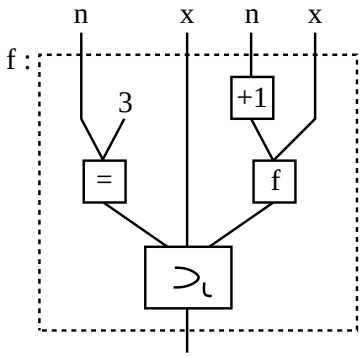

Figure 3: YF

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*.

5.3 Parallel extensions of PCF

The parallel extensions of PCF studied in Plotkin's seminal paper [18] are: *por*, pif_o (parallel conditional on booleans), and pif_ι (parallel conditional on integers). The functions are defined as follows:

$$\begin{array}{ll} por : o \to o \to o & pif_\sigma : o \to \sigma \to \sigma \to \sigma \\ por\ -\ tt = tt & pif_\sigma\ -\ x\ x = x \\ por\ tt\ -\ = tt & pif_\sigma\ tt\ x\ -\ = x \\ por\ ff\ ff = ff & pif_\sigma\ ff\ -\ x = x \end{array}$$

for $\sigma = \iota, o$. They are known to be extensionally equivalent [11, 20]. Interestingly, they are not intensionally equivalent.

Obviously, *por* can be used to implement n-ary disjunction as in CDSP, thus providing added intensional expressiveness over PCF. It turns out that pif_o and pif_ι can also be used for this purpose. However, *por* and pif_o cannot implement pif_ι efficiently and none of the constructs can speed up n-ary addition.

Proposition 5.7 *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.* [20]):

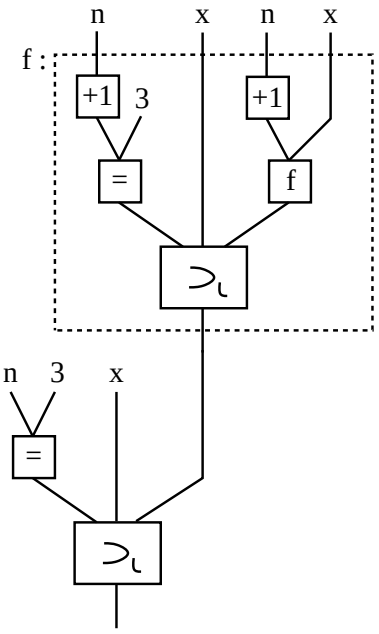

Figure 4: YF expanded once

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

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.* [20]):

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

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.* [20]):

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

This is clearly inefficient, because of the way the recursion unwinds, checking if x and y are equal to 0 first,

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

Table 1: Requirements for function B

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 5.8 *It is not possible to write a program in PCF + por, which computes $pif_\iota\ b\ x\ y$ and is of the form $\supset_\iota\ B\ x\ y$, 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:

1. If b is tt then B is tt,
2. If b is ff then B is ff,
3. If $(= x\ y)$ is tt then B is tt.

Table 1 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 Table 1 implies by monotonicity that $B\ ff\ tt = tt$. But this violates the monotonicity condition raised by the second line in the table. Therefore, no program of this form computes $pif_\iota\ b\ x\ y$. □

Our second lemma generalizes the first one.

Lemma 5.9 *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 "hard-wired" to always pick one of them. More precisely, we have two cases for the function computed by B:

1. 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 5.8.
   (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. What is the last gate in the one that evaluates to $pif_\iota\ b\ x\ y$? Apply the same reasoning here as in case (1b), eventually reaching case (1a).
2. 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. What is the last gate in N_1? We can apply the same reasoning here as in case (1b) of the proof, 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 5.8. 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 5.10 *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:

1. Static circuit. The result has type integer. Therefore, there are two cases for the last gate in the circuit:
   (a) $\supset_\iota$. By Lemma 5.9 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 5.9), and by the same argument the circuit cannot implement pif_ι.
2. 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 innermost recursion (there must be a constant number of them). 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. □

The next question we are concerned with is whether pif_ι is sufficient to 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.

We assume the existence of an addition operation $(+)$, as in CDSP, so we can write sequential addition without having to use recursion: $add_2 = \lambda xy.\ x + y$.

Proposition 5.11 *PCF + pif_ι cannot implement n-ary addition with a circuit of depth log n.*

Proof: We identify a property that holds for our CDSP 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 n. □

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

Proposition 5.12 *PCF + por cannot implement n-ary addition with a circuit of 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.

6 Conclusions

The sequentiality of the primitive recursive algorithms is manifested by their ability to recur on only one input. This makes them "ultimately obstinate," and they are not able to express an efficient algorithm for minimum.

The sequentiality of Berry-Curien algorithms is "by design." A sequential algorithm computes a sequential function, by only choosing one sequentiality index at a time, even if more than one exists. However, sequential algorithms are more expressive than primitive recursive algorithms: there is a sequential algorithm that computes a version of the minimum function efficiently, but not the "natural," inherently parallel, minimum function.

The addition of functional arguments to primitive recursion (system T) gives more power intensionally as well as extensionally. It allows us not only to express new functions, but also to compute more efficiently.

The addition of deterministic parallelism to CDS0 allowed us to compute the "natural" version of the minimum function, but CDS0 was already able to express an efficient minimum algorithm. However, the addition of deterministic parallelism did add intensional ex-

pressiveness, contradicting a conjecture from the literature. The computation of a number of functions can be speeded up, such as n-ary disjunction and n-ary addition. A more careful study of deterministic parallel constructs reveals different intensional powers. CDSP's query is more powerful than three parallel extensions to PCF, which differ in power among each other. Thus we have the beginnings of a hierarchy of intensional expressiveness for deterministic parallelism.

We have exhibited languages which are extensionally but not intensionally equivalent. The constructs *por*, pif_o, and pif_ι are interdefinable in the continuous function model of PCF. However, PCF + pif_ι 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.* [8]).

The study of intensional expressiveness has been neglected in the past, perhaps because it seems to be more difficult than extensional expressiveness. For instance, the problem of NC versus P can be phrased as a problem of relative intensional expressiveness between programming languages. Despite the difficulty, we believe that it is possible to obtain interesting results concerning intensional expressiveness, and that the area of intensional semantics deserves further exploration.

Acknowledgments

We thank Matthias Felleisen for many useful conversations on the topic of intensional expressiveness and for comments on an earlier version of the first part of the paper. We also thank the anonymous referees, whose suggestions led to improvements in the paper.

References

[1] G. Berry, Programming with concrete data structures and sequential algorithms, in: *Proc. ACM Conf. on Functional Programming Languages and Computer Architecture, 1981*, 49-57.

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

[3] G. Berry and P.-L. Curien, The kernel of the applicative language CDS: Theory and practice, in: M. Nivat and J.C. Reynolds, eds., *Algebraic Methods in Semantics* (Cambridge University Press, 1985) 35-87.

[4] S. Brookes and S. Geva, Computational Comonads and Intensional Semantics, in: M. Fourman, P. Johnstone, and A. Pitts, eds., *Applications of Categories in Computer Science*, LMS Lecture Notes 177 (Cambridge University Press, 1992) 1-44.

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

[6] R. Cartwright, P.-L. Curien, M. Felleisen, Fully abstract semantics for observably sequential languages, to appear in *Information and Computation*.

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

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

[9] T. Coquand, Une preuve directe du Théorème d'Ultime Obstination, *Comptes Rendus de l'Académie des Sciences*, March 1992.

[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.J. Gurr, Semantic Frameworks for Complexity, Doctoral Thesis, University of Edinburgh, Technical Report ECS-LFCS-91-130, January 1991.

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

[14] J. Hughes and A. Ferguson, A loop-detecting interpreter for lazy, higher-order programs, in: *Proc. Glasgow Workshop on Functional Languages, 1992*.

[15] G. Kahn and G.D. Plotkin, Concrete Domains, *Theoretical Computer Science* 121 (1993).

[16] S.C. Kleene, *Introduction to Metamathematics* (North-Holland, 1952).

[17] R. Milner, M. Tofte, and R. Harper, *The definition of Standard ML* (MIT Press, 1990).

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

[19] M. Rosendahl, Automatic complexity analysis, in: *Proc. ACM Conf. on Functional Programming Languages and Computer Architecture, 1989*, 144-156.

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