An Intensional Investigation of Parallelism (Thesis Proposal)

Denis Dancanet

September 21, 1994

Abstract

Denotational semantics is usually extensional in that it deals only with input/output properties of programs by making the meaning of a program a function. Intensional semantics maps a program into an algorithm, thus enabling one to reason about complexity, order of evaluation, degree of parallelism, efficiency-improving program transformations, etc. I propose to develop intensional models for a number of parallel programming languages. The semantics will be implemented, resulting in a programming language of parallel algorithms, called CDSP. Applications of CDSP will be developed to determine its suitability for actual use.

The thesis will hopefully make both theoretical and practical contributions: as a foundational study of parallelism by looking at the expressive power of various constructs, and with the design, implementation, and applications of an intensional parallel programming language.

1 Introduction

Denotational semantics has now been around for about 25 years, which makes it half as old as the computer itself. It is one of the three main approaches to providing semantics for programming languages (the other two being operational and axiomatic semantics). While it is a beautiful theory, which established a solid mathematical foundation for the enterprise of programming language semantics, it has not yet had much practical impact. There have been some applications [45], and concern for an elegant semantics has guided the design of some languages (Standard ML, Scheme), but it is usually the case that the language comes first, and then, perhaps, small pieces of it will be given a denotational semantics. Part of the reason for this is the fact that denotational semantics is a large and complex body of work; it is not terribly accessible. Another reason is that it is more suited for functional languages, which are essentially "sugared" versions of the λ-calculus. Modelling imperative features is possible, but it introduces a lot of complexity. The research proposed here can be seen as a step towards more practical applications.

Traditionally, denotational semantics has only been used to reason about *extensional* properties of programs. The meaning of a program is typically taken to be a function from input to output; there is no information about the way that function computes its result. For instance, two sorting programs, say bubblesort and mergesort, are mapped to the same input/output function by an extensional semantics, the function that sorts its input. However, the two programs are very different in terms of efficiency. This is an *intensional* feature. In an intensional denotational semantics, the meaning of a program is an algorithm embodying essential aspects of the computation strategy,

and thus bubblesort and mergesort can be differentiated. An intensional semantics enables us to reason about efficiency, complexity, order of evaluation, degree of parallelism, efficiency-improving program transformations, etc. Another use is in establishing relative *intensional expressiveness* results. Extensional expressiveness is concerned with whether a construct enables us to compute new functions. Intensional expressiveness is concerned with whether a construct enables us to write better programs. I will be referring to an intensional denotational semantics simply as an intensional semantics, although in general, an intensional semantics is any semantics which enables one to reason about intensional features.

Intensional semantics have (not surprisingly) been around for some time, in various fields, under various guises, but they have (surprisingly) been somewhat of a fringe research topic. This is probably due to the fact that it is easier to talk about extensional features. Now that the extensional theory is more mature, there is renewed interest in understanding intensional issues.

In my thesis, I propose to examine the theory and practice of intensional semantics in the setting of parallel programming languages. I intend to provide intensional models for a number of parallel programming languages. This will hopefully enable us to reason about intensional aspects of parallel programs and lead to a better understanding of the relative intensional expressive power of various parallel primitives. Then, viewing the semantics as a programming language of parallel algorithms (to be named CDSP), I intend to look at implementation issues and applications. The proposed work can be seen as an extension to the parallel world of Berry and Curien's work [3, 4, 5, 17] on an intensional semantics for sequential computation, which was implemented as a programming language of sequential algorithms (named CDS0).

The rest of this proposal is organized as follows: Section 2 describes in more detail what intensional semantics is, and why it is useful. Previous work that I will be directly building upon is described in Section 3. Section 4 presents some preliminary results and work I have done in the area. Section 5 describes the research proposed in this thesis. The expected contributions of the thesis are in Section 6, and the timetable for completion appears in Section 7. I conclude with related work in Section 8.

2 Intensional Semantics

The word *intension* is a loaded one in computer science in general, and even in the area of programming languages in particular. First, I give a brief description of the various usages of the word, pointing out the intended one in this proposal, followed by a discussion of intensional semantics proper, and a simple example.

2.1 The extension of intension

The word originated in philosophy: *intension* is the set of all attributes thought of as essential to the meaning of a term, as opposed to *extension*, which is the set of objects to which a term applies. It is used in logic: *intensional logic* is the branch of logic concerned with assertions whose meaning is dependent on an implicit context.

The logic usage led to one of the meanings in the programming languages community: an *intensional programming language* is one with context dependent constructs (for instance, a notion of execution time step). The first such language was Lucid, occurring in both sequential [21] and parallel [2] flavors.

Another meaning of intension/extension is the opposition between the function-as-a-program / function-as-a-graph views. It is used this way in recursion theory and proof theory, and it is the intended meaning here. An *intensional programming language* is one with constructs which make explicit intensional properties, such as order of evaluation, degree of parallelism, etc. An example of such a language is Berry and Curien's CDS0 [5].

2.2 What is intensional semantics?

Intensional semantics is a denotational semantics which makes the meaning of a program an algorithm, or more abstractly, a function paired with a computation strategy. An intensional semantics is more detailed than (and therefore is a refinement of) an extensional semantics.

Intensionality is relative: a semantics can be more intensional than another one. For each extensional semantics there is a hierarchy of intensional semantics that add more and more information. Our choice of an intensional semantics should be dependent on what program properties we wish to reason about, *i.e.,* we should be able to pick the relevant amount of detail for the problem at hand.

When one is not interested in the intensional aspects of program behavior, the intensional model should agree with the extensional one. In other words, one should be able to throw away the extra detail in an intensional model (*e.g.,* the computation strategy) and have it *collapse* onto an extensional model. If our intensional models are such "conservative extensions" of the extensional one, then we could reason about both intensional and extensional aspects at the same time.

Some of the intensional features and issues that can be reasoned about with an intensional semantics are (from [10]):

- Order of evaluation. Which argument does the program examine first? Which arguments are actually used? How much information about an argument is needed?
- Sequential versus parallel computation. Is the program sequential, parallel?
- Degree of parallelism. Which program is more parallel?
- Efficiency. What is the running time for a particular input?
- Complexity. What is the asymptotic running time of this program?
- Efficiency-improving transformations. Is this transformation extensionally correct *and* intensionally more efficient? Do we gain by parallelizing here?

In the past operational semantics has been used for some of these issues. Since operational semantics carries around with it a notion of abstract machine and low-level transitions on this machine (or in the case of structured operational semantics, a notion of transitions deducible from a set of rewrite rules), it typically retains too much detail and is unwieldy. It is possible, however, to discard some of the detail. An intensional *denotational* semantics still has some advantages: it is defined *compositionally* and permits algebraic reasoning (to show that two programs have the same meaning, we need only show that have the same denotation), and it enables the use of well-known techniques for reasoning about programs, such as fixed-point induction [45].

2.3 An example: Primitive recursion and the lazy natural numbers

As a simple example, we exhibit an intensional semantics for the primitive recursive ($\mathcal{PR}$) algorithms [33]. A $\mathcal{PR}$ algorithm can only recur on one input. The presentation of the algorithms is as a rewrite system following Colson [14].

Consider the following two algorithms for the addition of integers in unary representation (0, $S(0)$, ..., where S stands for successor) [15]:

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

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

The extensional denotational semantics for $add1$, $add2$ maps them into the same function, the addition function of type $N^2 \to N$, where N stands for the natural numbers. A simple intensional semantics is provided by the lazy natural numbers [14, 15, 16]. 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.

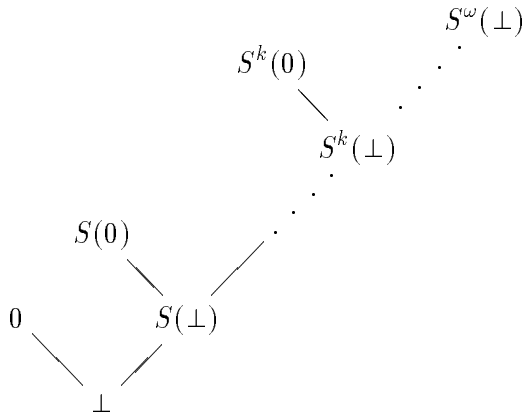

Figure 1: The lazy natural numbers

This simple intensional semantics is sufficient to distinguish between the two addition algorithms, $add1$ and $add2$. Using the meaning function $[\![\ ]\!]$ (from [15, 16]), mapping terms of arity n to functions of type $LNAT^n \to LNAT$, and which makes the meaning $\bot$ when an algorithm tries to recur on $\bot$, we have:

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

This is not particularly exciting, but this intensional framework can be used to reason about more interesting things, as Colson has done in his study of the expressibility of minimum. This is described in the next section.

3 Background

This section surveys the work that I will be building upon most directly in what follows. Section 3.1 gives a brief history of the full abstraction problem for PCF. Section 3.2 discusses Kahn and Plotkin's concrete data structures and formulation of sequentiality and its use by Berry and Curien in constructing an intensional semantics of sequential algorithms for PCF. Berry and Curien's programming language CDS0, which is a direct implementation of their intensional semantics, is described next, in Section 3.3. Sections 3.4 and 3.5 describe Brookes and Geva's extension of Berry and Curien's work to the setting of parallel algorithms and their formulation of a general theory of intensional semantics. Section 3.6 covers work on practical applications of sequential algorithms. Finally, Colson's work on the intensional expressiveness of primitive recursive algorithms is discussed in Section 3.7.

3.1 PCF and full abstraction

When a language is designed, the semantics which is normally regarded as "the definition" of the language is often presented in an operational style, by reference to the computations of an abstract machine. This leads to a notion of program equivalence based on observability: two programs will be considered equivalent if, when plugged into the same *context* (intuitively a program with a hole in it), we get the same final result.

If we give the language a denotational semantics as well, we obtain a different notion of program equivalence: two programs are equivalent if they denote the same value. It would be nice if these two notions of equivalence were the same. If that were the case we would say that the denotational semantics is *fully abstract* with respect to the operational semantics. The formulation is this way because the operational semantics is considered as given. A fully abstract semantics enables us to perform compositional reasoning.

Unfortunately, it turns out to be rather difficult to design fully abstract denotational semantics. In the setting of sequential languages, there has been a great deal of effort expended in inventing a fully abstract semantics for the language PCF (Programming Computable Functions) [25, 39]. PCF is regarded as the "prototypical" sequential programming language. It is a typed language with booleans, integers, primitive operations on integers, conditionals, and recursion.

Plotkin [39] showed that the normal denotational semantics (of cpos and continuous functions) for PCF is not fully abstract. The problem is that the denotational semantics is "finer" than the operational semantics. It makes too many distinctions. If two programs are denotationally equivalent, then they are operationally equivalent. The converse does not hold. This happens because the denotational semantics contains functions which are not definable in the language, like the parallel-or (*por*). *Por* returns true if at least one of its arguments is true:

$$por(tt, \bot) = tt$$
$$por(\bot, tt) = tt$$
$$por(ff, ff) = ff$$

Attempts to solve this problem have been aimed at eliminating the unwanted functions from the semantics, by restricting the continuous functions to "sequential" functions. Other ways of solving this are to change the operational semantics or to add primitives to PCF. Plotkin [39] showed that the denotational semantics for PCF + *por* (referred to as PCFP) is fully abstract. Cartwright,

Curien, and Felleisen showed [12, 17] that full abstraction can also be obtained by extending PCF with a *catch* primitive (referred to as PCFC).

Recently, Abramsky, Jagadeesan, and Malacaria [1] announced the solution of the original full abstraction problem for PCF. Their work involves game semantics, which can be seen as an elegant generalization of the work discussed in the next section.

3.2 Sequential algorithms on concrete data structures

Some of the early efforts at defining sequential functions were hampered by working in a setting where no distinction was made between function domains and domains of the data they compute on. In part to alleviate this, by providing a model in which it is easier to formalize the notion of incremental computation, Kahn and Plotkin [32] developed concrete data structures, and their domain-theoretic version, the concrete domains.

A concrete data structure (cds) is like a variant record in Pascal. It 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 filled with a value 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 is a partial order $\langle D(M), \subseteq \rangle$. A deterministic cds (dcds) is a cds satisfying two further conditions (well-foundedness and stability). The dcds are the ones used in what follows. Definitions can be found in [4, 17].

As an example, we can define the dcds of booleans ($BOOL$) the following way: there is one cell called B, which can be filled with either tt or ff. The set of states of this dcds is:

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

Using their cds's, Kahn and Plotkin defined a notion of sequential function. A continuous function f is sequential at some state x in its domain if for each cell c' accessible in $f(x)$ either (i) no cell is accessible in x, or (ii) there is an accessible cell c 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.

Berry and Curien [4] showed that Kahn-Plotkin cds's and sequential functions do not form a cartesian closed category (ccc), hence they cannot be used to model PCF. However, Berry and Curien defined *sequential algorithms* on cds's, which do form a ccc. This model is not useful for solving full abstraction for PCF because it is intensional (and it is known that the solution must be extensional), but that is exactly the feature of interest for this work; the meaning of a PCF term is an algorithm and the model is fully abstract with respect to a notion of observability that is sensitive to computation strategy. This is the first instance of an intensional model in the computer science literature that I am aware of.

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

Given two dcds's, M and M', the exponentiation dcds $M \Rightarrow M'$ 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.

For example, the state of $BOOL \Rightarrow BOOL$ that corresponds to the boolean negation is:

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

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.

3.3 The language CDS0

The programming language CDS0 [3, 4, 5, 20] is a direct implementation of the intensional denotational semantics presented above; hence, it is an intensional programming language of sequential algorithms. The name stands for Concrete Data Structures. The initial idea [3] was to make CDS0 a kind of "assembly" language for a (syntax-wise) ML-like language called CDS. Programs in CDS would be compiled down to CDS0. Only CDS0 was ever implemented [20].

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

- Uniformity of types. Everything in CDS0 is a state of a dcds. This can be a state-constant or a higher-order algorithm. The algorithm syntax is just syntactic sugar for the state of a dcds. 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 dcds corresponding to its type (hence a CDS0 object) is fully abstract with respect to two different operational semantics (CDS01 and CDS02) [17]. Since the semantics of an algorithm is a CDS0 object it is possible to write algorithms which manipulate the *semantics* of other algorithms.

Since CDS0 is supposed to be an assembly language, its syntax is rather heavy. I will restrain myself to just one example (more can be found in Curien's book [17]). The earlier definition of $BOOL$ is written in CDS0 and 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_STATE =
  {{}B     = valof B,
  {B=tt}B = output ff,
  {B=ff}B = output tt};
```

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

[1] This is only part of the definition: the full definition can be found in [17], for instance.

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 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.

A language in some ways similar to CDS0 has recently been developed by Cartwright, Curien, and Felleisen [12]. It is called SPCF (Sequential PCF) and it extends PCF with errors values and primitives for non-local transfer of control (catch and throw). This enables an SPCF program to observe and exploit order of evaluation in other programs. SPCF programs are called observably sequential algorithms. If there are no errors, SPCF collapses into CDS0. In the presence of errors, SPCF is an extension of CDS0.

3.4 Parallel algorithms on concrete data structures

Brookes and Geva [9] proceeded to extend Berry and Curien's work to the setting of deterministic parallel algorithms on concrete data structures. The aim was to provide a general intensional theory of deterministic parallel computation.

A parallel algorithm can also be viewed two ways. Abstractly, it is a pair of a *continuous* function and a (parallel) computation strategy. Concretely, it is a program in a language of parallel algorithms.

The key change to sequential algorithms to yield parallel ones is to replace the *valof* construct with a parallel *query* construct, which, intuitively, spawns off a number of *valofs*. More precisely, a *query* starts a number of parallel sub-computations and specifies conditions based on the results of the sub-computations under which the main computation may resume. As an example, here is how we could write parallel-or (of type $BOOL^2 \to BOOL$) in syntax meant to look like CDS0 as much as possible:

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

The previous algorithm, while deterministic, is a little misleading, because it seems to imply that one has knowledge of which argument evaluated, so one could write a non-deterministic program. To ensure determinism we have to make sure the output is the same for all *consistent* input states. Another problem is caused by higher-order parallel algorithms: their queries have to apply not only to the immediate input, but also possible further inputs. To account for all this, the notation would have to be somewhat different (see [9]).

Brookes and Geva were able to define application, currying and uncurrying of parallel algorithms. However, this only works for first-order types and composition was not defined. Thus, they did not obtain a ccc of parallel algorithms on concrete data structures.

It is important to note that Berry and Curien also had difficulty defining composition for sequential algorithms; their solution was to define it in terms of the abstract view of sequential algorithms.

3.5 A theory of intensional semantics

In related work, Brookes and Geva [7, 8] developed a more abstract theory of intensional semantics, formulated in the language of category theory. This work is a natural generalization of their *query* model, presented in the previous section. The basic idea is that given an extensional semantics we can construct many different intensional semantics from it. If the extensional meaning of a program is a function from values to values, then the intensional meaning is taken to be a function from computations to values, for some notion of computation that satisfies certain properties.

In their framework, computation is modelled as a comonad [35] with some extra structure. Depending on the comonad used in the construction, one obtains different notions of intensionality. For instance, one notion of computation is "increasing paths" over values. This essentially shows how a program behaves on a sequence of increasing information about its input, hence defining a computation strategy for it. That is why we can view the intensional meaning of a program as a pair of a function and a computation strategy, which we define to be an algorithm.

Brookes and Geva also explored the relationship between the extensional semantics and the intensional one constructed from it. If we start with a ccc as the extensional semantics and we have a comonad that preserves products, then the resulting intensional semantics is also a ccc.

3.6 Applications of sequential algorithms

A sequential algorithm contains detailed information about the relationship between input and output. It doesn't just tell us how the output depends on the input, but precisely how *parts* of the output depend on *parts* of the input. Seizing on this intensional information, Hughes et al. [22, 29] developed applications using sequential algorithms as a representation for functional programs. In [29] a loop-detecting interpreter for a lazy, higher-order language is described. The standard approach to detecting loops is to check for a recursive function being called twice with the same arguments. But this doesn't work for higher-order functions. Using sequential algorithms we can get around this problem. The knowledge of what input cells a particular output cell depends on enables us to find the fixpoint of a sequential algorithm more efficiently. By detecting self-dependent parts of the fixpoint we find loops. Of course, not all loops can be detected, as this is an undecidable problem.

In a later paper [22], an abstract interpretation based on sequential algorithms is developed for a higher-order language. Again, the gains come from the intensionality of the representation. The resulting interpretation is faster than other approaches.

3.7 Colson's work on intensional expressiveness

We could define the minimum of two integers in unary representation in a natural way by the following rewrite system:

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

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

Note that this is not a primitive recursive ($\mathcal{PR}$) algorithm; there is simultaneous recursion on two inputs.

We need to distinguish between the function min defined above, and the algorithm min. Intuitively, by applying the rewrite rules, the algorithm $min(n, p)$ computes its result in $O(min(n, p))$ time (it takes exactly $min(n, p) + 1$ steps).

Colson studied the expressibility of minimum in the context of $\mathcal{PR}$ algorithms [14, 15]. He established that $\mathcal{PR}$ algorithms are inherently sequential: they possess sequentiality indices. Using primarily the intensional semantics of lazy natural numbers ($LNAT$), which we exhibited earlier, he 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 which computes the minimum of two numbers n and p in unary representation, and is of complexity $O(min(n, p))$.*

However, there are many $\mathcal{PR}$ algorithms which compute the minimum of two integers. We define one below, using some auxiliary functions (see [33]):

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

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

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

Note that in an operational interpretation of this definition, the algorithm $MIN(n, p)$ has a worst-case running time of $O(max(n, p))$. Let us call the elements of $LNAT$ of the form $S^k(0)$ *defined*, and the elements of the form $S^k(\bot)$ *partial*. The function MIN agrees with min from the previous section on the defined elements of $LNAT$. They are different on the partial elements. By 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 result: $\mathcal{PR}$ algorithms can compute MIN but not min. Note that there are many other functions between min and MIN. 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 [24]. In system T we can not only compute new functions (*e.g.,* Ackermann function), but we can also compute minimum efficiently (system T can express an algorithm for min [14]). System T is more powerful than $\mathcal{PR}$ extensionally and intensionally.

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

4 Preliminary Results and Research

4.1 Intensional expressiveness

I have begun an investigation [11] of the intensional power of CDS0, and of the relative intensional expressiveness of various deterministic parallel primitives.

4.1.1 CDS0

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

It is easy to show that *min* cannot be computed by a sequential algorithm, because it is not a sequential function[2]:

Proposition 4.1 *There is no sequential algorithm which computes min.*

But this does not mean we cannot compute minimum efficiently in CDS0. The problem with $\mathcal{PR}$ algorithms is that they suffer from "ultimate obstination" [15, 16]: at some point one argument must be chosen to be evaluated until the end. Sequential algorithms allow us to alternate between the two inputs, examining one cell at a time.

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

The CDS0 algorithm for minimum does not denote *min*, because that function is not sequential. The difference between the denotations of the CDS0 minimum (called *left_min*) and *min* is illustrated by their behavior on inputs of the form $(S^n(0), S^n(\bot))$ (they agree on all other inputs):

$$[\![left_min]\!](S^n(0), S^n(\bot)) = S^n(\bot) \qquad [\![min]\!](S^n(0), S^n(\bot)) = S^n(0)$$
$$[\![left_min]\!](S^n(\bot), S^n(0)) = S^n(\bot) \qquad [\![min]\!](S^n(\bot), S^n(0)) = S^n(0)$$

The addition of the *query* construct enables us to compute *min*. It is essentially the same situation as computing *parallel-or*. So we have the following:

Proposition 4.3 *There is a CDS0 + query program which computes min.*

4.1.2 Deterministic parallelism

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. Does deterministic parallelism provide added intensional expressiveness? There appears to be a folk conjecture that deterministic parallelism is not "useful." The claim (see [12], for instance) is that even though deterministic parallel features increase the extensional expressiveness of a language,

[2] Proofs are omitted in this section; refer to [11] for details.

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 is extensionally, but not intensionally more expressive.

This conjecture is false. Deterministic parallelism does provide added intensional expressiveness. The deterministic *query* construct is sufficiently general to allow a speed-up in the computation of many different functions. Three parallel extensions to PCF were also studied: 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.

Using *query* we can compute various n-ary functions, such as n-ary or, n-ary addition in $O(log\ n)$, by constructing a balanced tree of processes. These functions can only be computed in $O(n)$ in CDS0. So we have:

Proposition 4.4 *CDS0 + query is intensionally more expressive than CDS0.*

The three parallel extensions to PCF can all be used similarly to speed up the computation of n-ary or, so they are also intensionally more expressive than PCF. *por* and pif_o can be implemented in terms of each other in constant time, and pif_ι can implement the other two in constant time. Interestingly, however, pif_ι cannot be implemented in terms of *por* or pif_o in constant time.

Proposition 4.5 *PCF + por cannot implement pif_ι in constant time.*

The problem is that pif_ι can start parallel subcomputations on integers, whereas *por* (pif_o) can only do it on booleans. The proof is by contradiction, and involves viewing a PCF program as a circuit and arguing based on the last gate used in the circuit.

But even pif_ι is not as expressive as query, since it cannot compute n-ary addition in $O(log\ n)$. The reason is that pif_ι has a limited kind of parallelism: it can evaluate two integers in parallel, but it must return one of them.

Proposition 4.6 *PCF + pif_ι cannot implement n-ary addition in $O(log\ n)$.*

In light of these results, we have the emergence of a picture of different levels of intensional expressiveness for deterministic parallel constructs.

4.2 CDS0 implementation

I have implemented an interpreter for CDS0 [18]. It is implemented in Standard ML/NJ and is based on Devin's thesis [20].

Berry and Curien devised two operational semantics for CDS0: CDS01 and CDS02 [5, 17]. When the user types a cell name in the request loop, that cell name can be higher-order (*i.e.,* it can have a state part). In CDS01 these states must be listed explicitly, whereas in CDS02 expressions may be used. We say that CDS02 allows *intensional queries*. This makes it "lazier" and more efficient than the CDS01 semantics, except in the evaluation of recursion, where CDS01 is faster because it is essentially memoized (it maintains internal tables of state approximations). CDS02 also imposes a restriction on the cds: they must be sequential [17], but this restriction is inessential in practice. My interpreter uses the CDS02 operational semantics.

The performance of the implementation has been acceptable on the relatively small examples tried so far, even though no particular attention was paid to efficiency issues.

I am in the process of developing a type inference algorithm for CDS0. An implementation is already working, but it produces an approximation of the type. I provide a brief description here.

Cds's can be seen as generalizations of records, but the meaning of a type is different from the record world [25]. A record $\{N = 7\}$ has type $\{N{:}int\}$, and this type is true of all records having such a field (they could have more). If we add another field, the resulting type is a subtype of $\{N{:}int\}$ because it holds for fewer records. The situation is roughly reversed in the cds world. Consider the cds *SEVEN* with a cell N which can be filled with the value 7, and the cds *INT* with a cell N which can be filled with any integer. The only states with type *SEVEN* are $\{\ \}$ and $\{N = 7\}$. In contrast, more states have type *INT*. So *SEVEN* should be a subtype of *INT*.

This notion of subtyping ($\leq$) is related to the notion of inclusion in [17]. Intuitively, $M_1 \leq M_2$ if M_2 has all the cells and values per cell that M_1 has, and M_2's access conditions are "weaker." This definition leads to a covariant subtyping rule for arrow:

$$\frac{\sigma_1 \leq \sigma_2 \quad \tau_1 \leq \tau_2}{\sigma_1 \to \tau_1 \leq \sigma_2 \to \tau_2}$$

Deciding subtyping on infinite cds's is equivalent to inclusion on infinite regular trees, hence is decidable. Even in the presence of subtypes, an algorithm will, in general, yield an intersection type (there can be incomparable cds's with the same cell names and values that the algorithm refers to). The current type for an algorithm is an approximation (we choose the highest type that holds for the algorithm, given the type hierarchy). Polymorphic types (algorithms defined with generic cell and value references) are supported in the current implementation.

More work needs to be done in understanding if more information about the type of an algorithm can be conserved. It might be possible that no type inference is possible. In that case, it would be worthwhile to develop a type-checking system. Nevertheless, the current implementation, even with its limitations, has been very useful in making the task of writing CDS0 programs easier.

5 Proposed Research

I propose to investigate theoretical and practical aspects of generalizing Berry and Curien's sequential algorithms to parallel algorithms. This would result in an intensional semantics for parallel languages. Just as Berry and Curien's work with CDS0 yielded insight into the nature of sequential computation, I hope that an intensional study of parallelism would provide a deeper understanding of parallel computation.

Note that parallel here refers to the theoretical sense of the word; a parallel language is one which allows the expression of parallel functions, such as *parallel-or*. Data-parallel languages, for example, are not parallel in this sense, since the programs they compute have sequentiality indices (all subparts of the program must complete in order for the program to return a result).

There are many possible lines of research in this direction:

1. First of all, there are several ways of generalizing sequential into parallel algorithms. I have begun an investigation of Brookes and Geva's deterministic *query*. I would like to study others as well, and prove relative intensional expressiveness results.

2. There should be a semantic theory for parallel algorithms that would allow us to reason about intensional aspects.

3. Given a language of parallel algorithms, it should be used as the intensional semantics of parallel programs. The translation of programs from various parallel languages to CDSP should be studied.

4. The work on abstract interpretation based on sequential algorithms opens the door for trying to exploit the same kind of intensional information to produce an abstract interpretation for parallel programs based on parallel algorithms.

5. Finally, a great deal of implementation work is needed. There should be a higher-level syntax for CDS0 and some of the parallel extensions to it should be implemented.

I describe the various lines of research in more depth in what follows. Note that they are closely related, so there is some overlap between the various sections.

5.1 The family CDSP

Parallel generalizations of CDS0 fall into two broad categories: deterministic and non-deterministic. I have already begun a study of deterministic *query* and obtained some partial results. The interest in studying non-deterministic extensions stems from the apparently widespread belief that non-determinism is more "powerful." The expectation is that non-deterministic extensions are intensionally more expressive than the deterministic versions. Another reason for studying non-determinism is that most existing parallel languages are non-deterministic.

Aside from deterministic *query*, I intend to study at least the following parallel generalizations of CDS0:

- Non-deterministic *query*,
- Communicating sequential algorithms.

The questions I am interested in are:

- How do they extend CDS0? What can be computed faster?
- Is there some generalization that is intensionally more expressive than the others?

The second generalization, deserves some comment. The intent is to be able to model a language like CSP [28], or CML [40]. Initially, no dynamic process or channel generation will be allowed, but that should be relaxed. The difficult part will probably be modeling the channels. One solution is to make a channel an infinite cds of like cells; using the channel once enables the next channel cell to be used. This way, the original CDS0 property that a cell is not filled or visited more than once would be preserved.

Since every project must have an acronym, the family of parallel generalizations of CDS0, unrelated though they may be, will be known, collectively, as CDSP (CDS Parallel).

5.2 An intensional model for parallel algorithms

A second aspect of the study of CDSP is the construction of a model for it: an intensional model for parallel algorithms. This is where reasoning about CDSP programs will be carried out.

As we have seen, the early efforts of Brookes and Geva [9] were unsuccessful, as they could not get a category. However, their comonad semantics [7, 8] can be used to construct a category of parallel algorithms. The "increasing paths" comonad is product preserving, thus giving rise to a ccc, but it contains many parallel algorithms which are not denotable by any CDSP term, given the generalizations of CDS0 currently envisioned; thus, it is unlikely that the model will be fully abstract. The "strictly increasing paths" comonad removes a lot (all?) of the non-denotable algorithms, but is not product preserving, and does not give rise to a ccc. This is not necessarily a problem—an intensional model might not have to be a ccc. It remains to be seen if the comonad semantics is the right choice in this context, or if a new kind of intensional model of parallel algorithms needs to be developed.

Given a model, I intend use it to reason about efficiency, efficiency-improving program transformations, degree of parallelism, and so on, of CDSP programs.

5.3 CDSP as intensional semantics for parallel languages

A third aspect of the study of CDSP is to use it as the intensional semantics of other parallel programming languages. The idea is to translate programs from other languages to CDSP with the goal of extracting useful intensional information and proving relative intensional expressiveness results.

I plan to use CDSP to model at least two languages:

- PCF Parallel (PCF + *por*, pif_o, pif_t).
- A subset of Concurrent ML, or some other CSP-like language.

The translation of PCFP to CDSP should be fairly trivial. The translation of PCF to CDS0 has been studied extensively, being none other than the translation of a functional language to the *categorical combinators* [17]. In the intensional expressiveness results obtained so far about PCFP, I have not made use of CDSP.

Modeling a higher-order parallel language based on send/receive primitives should also be fairly simple. As mentioned, the main problem might be the channels.

A very distant goal is to consider the semantics of *futures*, from languages such as MultiLisp [27]. A future is a promise to provide the value of a computation, if needed. The computation can proceed in parallel and pointers to the future can be manipulated. It is very difficult to determine where to put futures in a program in order to achieve the maximum amount of useful parallelism. Perhaps intensional semantics can be used for this purpose.

5.4 Abstract interpretation

Another possible line of work is to extend the work on abstract interpretation based on sequential algorithms to parallel programs. Using the intensional representation of parallel algorithms it should be possible to obtain analyses of higher-order parallel programs. I would like to consider some of the following properties:

- Data dependence. This is one of the analyses that are usually attempted for parallel programs [13, 31], although with an imperative flavor. The parallel algorithm representation should be ideally suited for this.

- Extra parallelism. We could find out if certain arguments are needed (strictness analysis) and if they can be evaluated in parallel.
- Selective communication. In a language with channels and send/receive primitives, a process might have to keep track of several channels on which it could receive an input. It would be useful to know if no message will come on a particular channel.
- Deadlock detection. Using the data dependence information it might be possible to determine deadlock.

The various abstract interpretations should be implemented. Based on the experience of Hughes et al. [22] there is hope that the implementation will be fast.

5.5 Implementation and applications

Another aspect of the study of CDSP is the practical aspect: implementation and applications.

I will extend my CDS0 interpreter with various parallel primitives. Initially, I will implement *query*. The implementation will be either in SML using continuations, or in CML [40], which provides the possibility of spawning processes, as well as typed, named channels. It is unlikely that much effort will be expended in the direction of an efficient implementation. It is possible that I will incorporate some of the improvements studied by Hughes et al. [30] in their quest for a more efficient interpreter. I expect, however, that the implementation will be more of a proof-of-concept.

CDSP should have a higher-level syntax than CDS0. From experience programming in CDS0, this seems a necessity. I have developed an SML-like notation for CDS0 programs, but it is not yet complete, and it is not clear how hard the translation to the lower-level language will be. It might be better to dispense with the requirement of being able to write intensional programs at the higher level, and simply use a generic lazy, higher-order language. Another possibility which retains the intensional ability is Felleisen's PCFC [12].

In the interest of usability, it would be nice if CDSP had type inference. This is unrelated to the theoretical aims of the thesis but is a consideration from the point of view of practicality. It should be fairly easy to change a CDS0 type inference or type-checking algorithm for CDSP.

6 Expected Contributions

The expected contributions of this thesis are both theoretical and practical:

- A study of the intensional expressive power of various parallel primitives.
- A theory of intensional semantics for parallel programs.
- Intensional models for a variety of parallel languages, including PCFP, and a subset of Concurrent ML.
- An implementation of the intensional parallel programming language CDSP.
- Applications of CDSP with regards to the analysis of intensional properties.
- A type-inference or type-checking system for languages based on cds.

7 Timetable

The timetable for completion of the thesis (in non-tabular form) follows. Items are not necessarily listed in chronological order. Time is given in square brackets in months. The expected completion date is December 1995.

- Explore various extensions to CDS0 [1].
- Find a good intensional model for CDSP [2].
- Examples of using the model to reason about intensional issues [2].
- Concurrent ML and related issues [1].
- CDSP implementation [2].
- CDSP applications and evaluation [2].
- Finish work on type inference for cds [1].
- Writing the thesis [4].

8 Related Work

The related work surveyed here is composed of several different strands. The common element is a concern with the analysis of intensional aspects of programs. In most cases, the programming language is sequential, and the analysis is carried out from an operational presentation of the semantics. The notable exceptions are: Zimmermann [49, 50] who studies automatic complexity analysis for a deterministic parallel language and Gurr [26] who extends denotational semantics to reason about intensional aspects of first-order sequential languages.

I discuss some of the work in the following topics: intensional semantic models for programming languages, intensional hierarchies, and automatic complexity analysis. The section is broken down by area (*e.g.*, recursion theory), rather than topic, to give an idea of the naturality and pervasiveness of these ideas.

8.1 Recursion theory

The distinction between a function and the algorithm that computes it was made early on in recursion theory [41], but the main focus of the theory is on the functions, that is on the extensional features. Most results have to do with closure properties of various collections of recursive functions.

However, a so-called "abstract" recursion theory (also called theory of algorithms) has been developed, chiefly by Moschovakis [36, 37, 38], although the ideas go back to Kleene and others. In [36] Moschovakis develops the foundation for the theory. The semantics of a recursive partial function is a set of functionals, called a *recursor*. It is essentially a higher-order functional program defined by mutual recursion. Intensional analysis can be performed in an operational style on the recursor. The possibility of implementing the language of recursors as a programming language called REC is discussed. More recent works [37, 38] update and elaborate on the older paper. Algorithms are modelled as recursors, which are part of a programming language called FLR

(Formal Language of Recursion). The main thrust is in proving that FLR is a reasonable language in terms of including all desirable intensions.

8.2 Proof theory

Proof theory [23, 24] has been mainly concerned with extensional aspects, as well. A series of functional systems of increasing *extensional* expressive power has been studied: linear λ-calculus, typed λ-calculus, primitive recursion, Gödel's system T, Martin-Löf's intuitionistic type theory, Girard-Reynolds polymorphic second-order λ-calculus (system F), and the theory of constructions. None of these systems is Turing-complete; all their programs terminate. But in a sense, these results are not that relevant for computer science, since non primitive recursive functions are intractable. Do we gain anything by going to a more powerful system? Can we write better algorithms?

Recently there has been work on intensional aspects of some of these functional systems in an attempt to settle such questions. Colson's work [14, 15] with primitive recursive algorithms was mentioned already. He also studied system T and system F. System T can express an efficient algorithm for minimum. It is an open problem whether $min(n,p)$ can be written in system F with complexity $O(min(n,p))$. The current best program (see [19]) is $O(min(n,p)log(min(n,p))$. Interestingly, system T appears to be intensionally stronger than system F, even though it is extensionally weaker.

8.3 Programming languages

There is a large body of literature devoted to automatic complexity analysis. The first important point to note about this endeavor is that it is undecidable. The second point is that it is quite complicated to analyze certain algorithms. Therefore, an automatic analysis system will not work all the time and, to be complete, must contain an implementation of the state of the art in algorithm analysis techniques.

There are two phases to an automatic complexity analysis system: deriving recurrences for the complexity of a program, and solving them. Deriving the recurrences is usually done by constructing a cost (or complexity) function from the program and finding a function of the input size from it. The cost function normally counts the number of rewrites in the operational semantics, plus constants for the primitive operations.

With access to an intensional denotational semantics, it would be possible to use the denotation of a program to derive the recurrences. There is probably not much to be gained from doing this, however, as the key part of automatic complexity analysis is solving the recurrences.

Most of the work in automatic complexity analysis has been devoted to studying strict, sequential, first-order, functional languages (the earliest examples are Wegbreit [48] and Le Metayer [34]). There has been some effort in the area of lazy first-order languages [47, 44]. The derivation of a complexity is more complicated in this this setting, because only part of an argument might be needed. There has also been work with higher-order strict and lazy languages [43]. The basic idea is to construct *cost-closures* so a function can carry around cost information.

I am aware of only one example in the literature involving a parallel language. Zimmermann [49, 50] analyzes a first-order parallel language with vectors and a parallel "for-all" construct (and similar others) ranging over vectors. The approach is the same as in the sequential case: a cost function is constructed, with the cost of the parallel "for-all" equal to some constant plus the *maximum* cost of the operation over each vector element.

An interesting approach to automatic complexity analysis, in the setting of strict, sequential, first-order languages, was taken by Rosendahl [42]. He also constructs a time (or cost) function from the program, but in order to talk about the correctness of this time function, he defines an "instrumented" denotational semantics which returns a denotation and the time complexity. A *time-bound* function, which gives an upper bound on computation time for all inputs of a certain size, is derived by abstract interpretation from the time cost function.

Talcott developed a theory of intensional semantics [46]. Essentially, the extraction of the intensional information is based on a low-level operational semantics: from a program she constructs a *computation sequence*. Analysis is performed on the computation sequence: the time complexity of a program is the length of its computation sequence. Other properties can be analyzed, such as maximum stack depth and number of function calls.

Gurr [26] extended denotational semantics in order to model intensional aspects (resource requirements) of first-order, sequential languages. In his framework, the meaning of a program is a pair of the original denotation of the program and a map from input values to an object of resource values. The object of resource values is modelled as a *monoid* (a semigroup with identity). Time and space requirements of programs can be formulated in this framework. He also studied the derivation of exact and non-exact complexities.

References

[1] S. Abramsky, R. Jagadeesan, P. Malacaria, Games and full abstraction for PCF II: Second preliminary announcement, 1993.

[2] E.A. Ashcroft, A.A. Faustini, R. Jagannathan, An intensional parallel processing language for applications programming, Technical Report SRI-CSL-89-1, SRI International, 1989.

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

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

[5] 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.

[6] G. Berry, P.-L. Curien, J.-J. Lévy, Full abstraction for sequential languages: the state of the art, same source as [5], 89-132.

[7] S. Brookes and S. Geva, Computational Comonads and Intensional Semantics, Technical Report CMU-CS-91-190, Carnegie Mellon, 1991.

[8] S. Brookes and S. Geva, Continuous functions and parallel algorithms on concrete data structures, in: *Mathematical Foundations of Programming Semantics, Pittsburgh, 1991* (Springer LNCS 598, 1991) 326-349.

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

[10] S. Brookes, A Theory of Intensional Semantics, lecture at Carnegie Mellon, Spring 1993.

[11] S. Brookes and D. Dancanet, Sequential Algorithms, Deterministic Parallelism, and Intensional Expressiveness, to appear in POPL '95.

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

[13] J.-H. Chow and W.L. Harrison III, Compile-time analysis of parallel programs that share memory, in: *Proc. Principles of Programming Languages, 1992*, 130–141.

[14] L. Colson, About primitive recursive algorithms, in: G. Ausiello et al. eds., *Proc. 16th International Colloquium on Automata, Languages and Programming* (Springer-Verlag LNCS 372, 1989), 194-206.

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

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

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

[18] D. Dancanet, CDS0 User's Guide (version 1.0).

[19] R. David, The Inf function in the system F, manuscript.

[20] M. Devin, *Le Langage CDS: Description, Implémentation, Compilation*, Thèse Docteur Ingénieur, Université Paris VII (1984), Rapport LITP 85-13.

[21] A.A. Faustini and W.W. Wadge, Intensional programming, in: J.C. Boudreaux et al. eds., *The Role of Language in Problem Solving 2*, (North-Holland, 1987), 119-132.

[22] A. Ferguson and J. Hughes, Fast abstract interpretation using sequential algorithms, in: *Proc. Padova Workshop on Static Analysis, 1993*.

[23] J.-Y. Girard, *Proof Theory and Logical Complexity I* (Bibliopolis, 1987).

[24] J.-Y. Girard, Y. Lafont, P. Taylor, *Proofs and Types* (Cambridge Tracts in Theoretical Computer Science 7, 1990).

[25] C.A. Gunter, *Semantics of Programming Languages* (MIT Press, 1992).

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

[27] R.H. Halstead, MultiLisp: A language for concurrent symbolic computation, in: *ACM Transactions on Programming Languages and Systems*, 7(4):501-538, October 1985.

[28] C.A.R. Hoare, Communicating Sequential Processes, in: *Communications of the ACM* 21 (8) 1978, 666–677.

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

[30] J. Hughes, S. Hunt, C. Runciman, Higher-order functions as decision trees: Taming a space monster, 1994, manuscript.

[31] S. Jagannathan and S. Weeks, Analyzing stores and references in a parallel symbolic language, in: *Proc. Lisp and Functional Programming, 1994*.

[32] G. Kahn and G.D. Plotkin, Concrete Domains, in: *Theoretical Computer Science* 121(1993). Earlier available in French as: Domaines Concrets, IRIA Report 336, 1978.

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

[34] D. Le Metayer, Mechanical Analysis of Program Complexity, in: *ACM SIGPLAN Symposium on Language Issues in Programming Environments, Seattle, 1985* 69-73.

[35] S. MacLane, *Categories for the Working Mathematician* (Springer-Verlag, 1971).

[36] Y. Moschovakis, Abstract recursion as a foundation for the theory of algorithms, in: M.M. Richter et al. eds., *Computation and Proof Theory* (Springer-Verlag LNM 1104, 1984), 289-364.

[37] Y. Moschovakis, The Formal Language of Recursion, *The Journal of Symbolic Logic*, vol. 54, 1989, 1216-52.

[38] Y. Moschovakis, A mathematical modeling of pure, recursive algorithms, in: A. Meyer and M.A. Taitslin eds., *Logic at Botik '89: Symposium on Logical Foundations of Computer Science* (Springer-Verlag, 1989), 208-29.

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

[40] J. Reppy, CML: A Higher-Order Concurrent Language, revised version of paper presented at *SIGPLAN Conf. on Programming Language Design and Implementation, 1991*, 1993.

[41] H. Rogers, Jr. , *Theory of Recursive Functions and Effective Computability* (MIT Press, 1987).

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

[43] D. Sands, Complexity analysis for a lazy higher-order language, in: *Proc. Third European Symposium on Programming, 1990*, 361-76.

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

[45] D.A. Schmidt, *Denotational Semantics* (Allyn and Bacon, 1986).

[46] C. Talcott, Rum: An intensional theory of function and control abstractions, in: *Workshop on Foundations of Logic and Functional Programming, Trento, 1986* (Springer, 1988) 3-44.

[47] P. Wadler, Strictness analysis aids time analysis, in: *Proc. Symposium on Principles of Programming Languages, San Diego, 1988*, 119-32.

[48] B. Wegbreit, Mechanical Program Analysis, *Communications of the ACM* 18(9) 1975, 528-39.

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

[50] W. Zimmermann, The automatic worst case analysis of parallel programs: simple parallel sorting and algorithms on graphs, Technical Report ICSI 91-045, International Computer Science Institute, 1991.