Operational Semantics Models of Complexity
(Thesis proposal)

John Greiner

April 22, 1994

Abstract

Definitions of complexity are traditionally given relative to a relatively low-level, machine-oriented model of computation. As programming languages become more complex, the relation between a program and its machine model time, space, *etc.*, complexities becomes less clear. An alternative is to provide a language-oriented definition of complexity in terms of costs that make sense for that language and its semantics. The language-level complexity can then be related to the machine-level complexity. The programmer can use the more intuitive language-level definition and then use this relation, which is proved separately, to obtain machine-oriented complexities when needed.

1 Introduction

Definitions of complexity must be given relative to a model of computation. Traditionally, these models have been machine models, such as the Random Access Machine (RAM) and Turing Machine for serial computation, and the Parallel RAM (PRAM), circuits, and models based on specific communication networks for parallel computation. To provide a higher level of abstraction, it is common to use first-order Algol-esque language with an informal argument that this can be mapped simply to the machine model, *e.g.*, Pidgin Algol in [1].

But as programming languages become more complex, the relation between a program and a machine model becomes less clear. Costs of language primitives are not necessarily constant and can depend on the values or sizes of arguments. As a result, the costs involved in these languages may not be clear to the programmer. Some examples of more complex features in modern languages are first-class functions, environments, continuations, and automatic storage management.

Furthermore, what underlying machine model is appropriate might not be known to the programmer, as languages can be implemented on radically different architectures. Deriving complexity results is particularly a problem when the language is implemented on multiple parallel architectures or on serial and parallel architectures.

An alternative approach is to define a cost model directly for the language. For the pseudocode used in practice, this amounts to formalizing what is traditionally done informally. Also, this generalizes the traditional method, since the distinction between "languages" and "machines" is informal and only for explanatory purposes.

From the perspective of programming language semantics, the cost information allows finer distinctions to be made between languages. For example, the call-by-need pure λ-calculus is semantically equivalent to the call-by-name pure λ-calculus. But with cost information, their differ-

ence in complexity can be defined and studied. Similarly, with languages such as NESL [5], cost models can express the parallelism intended for their implementation even though their semantics are sequential. These cost models show the difference between the intended implementations and ones defined using serial languages, for example.

Language-based models should be useful for several reasons:

- They formalize and extend the use of (realistic) programming languages as the basis for complexity analysis.
- Finer distinctions can be made between languages by using cost information.
- A programmer can determine the complexity of a program relative to the language without knowledge of the machine model. Separately, this complexity can be related to the machine model.
- Determining the complexity of a program relative to the language lends itself to automation.
- Efficiency results at the language level can hold at the machine level, as well. In particular, I intend to show that for some models, a program that is work-efficient relative to the language-based model is also work-efficient relative to the machine-based model. Part of this investigation will be to discover conditions on models for this to hold.

Part of this dissertation will be comparing the *cost-expressiveness* of language complexity models, particularly for parallel languages. Informally, a language complexity model is more cost-expressive than another if the first model allows problems to be solved with lower complexity. This is similar to, but not directly related to, other notions of expressiveness [9, 19].

2 Profiling semantics

The basis of my language-based definition of complexity will be a *profiling semantics*. A profiling semantics is an operational semantics augmented to reflect the costs incurred by evaluating an expression. While cost-augmented semantics have been used before, they have not been used for this general of a purpose (*cf.* Section 6).

I use an operational, rather than denotational, semantics as a basis for two subjective reasons. First, I am assuming that programmers typically use the operational semantics (formally or informally) as their basis of understanding the behavior of a programming language. Also, it seems more appropriate to add cost information to a semantics that is intended to describe the behavior of the language, rather than just the result of a computation.

As an illustrative example, I will use a model based on the call-by-value λ-calculus. Traditional complexity theory is based on well-defined machines, with informal arguments extending practice to simple while-loop-based languages similar to Fortran or C. But these informal arguments are not complex enough to describe many language features such as higher-order functions, continuations, or objects.

The call-by-value λ-calculus is a simple language which forms the core of some modern functional languages, such as Scheme and ML. This example is only meant as an explanatory tool.

The syntax of this language is

$$x \in \mathit{Variables}$$

$$
\begin{array}{rcl}
i \in \mathit{Integers} & & \\
k \in \mathit{Constants} & ::= & i \mid + \mid - \mid \times \mid /2 \mid \mathit{pos?} \\
e \in \mathit{Expressions} & ::= & k \mid x \mid \lambda x.e \mid e_1\ e_2 \\
v \in \mathit{Values} & ::= & k \mid +_i \mid \times_i \mid \mathit{Clos}(E, x, e)
\end{array}
$$

where an environment E is a finite mapping from variables to values. An empty environment is denoted $[]$, and the extension of an environment is denoted $E[x \mapsto v]$. The symbol $-$ denotes negation, not subtraction.

This includes some basic arithmetic constants to allow reasonably sized examples. However, full division is not included, so that the integers are incompressible [3]. This means that it is not possible to encode an unbounded number of integers in a single integer with only constant overhead. This ensures that encoding tricks do not result in more efficient algorithms. Adding full division for practicality can only improve potential complexity bounds.

The operational semantics of the language is defined by the relation, or judgment, $E \vdash e \longrightarrow v$, which is read "In the environment E, the expression e evaluates to the value v." Operational semantics are commonly defined with natural deduction rules of the form

$$\frac{\text{Assumptions}}{\text{Conclusion}} \qquad \text{(RULE NAME)}$$

So, the evaluation judgment is the smallest relation provable from the following standard rules.

$$\frac{E(x) = v}{E \vdash x \longrightarrow v} \qquad \text{(VAR)}$$

$$E \vdash k \longrightarrow k \qquad \text{(CONST)}$$

$$E \vdash \lambda x.e \longrightarrow \mathit{Clos}(E, x, e) \qquad \text{(LAM)}$$

$$\frac{E \vdash e_1 \longrightarrow \mathit{Clos}(E', x, e) \qquad E \vdash e_2 \longrightarrow v_2 \qquad E'[x \mapsto v_2] \vdash e \longrightarrow v}{E \vdash e_1\ e_2 \longrightarrow v} \qquad \text{(APPLY)}$$

$$\frac{E \vdash e_1 \longrightarrow v_1 \qquad E \vdash e_2 \longrightarrow v_2 \qquad \delta(v_1, v_2) = v}{E \vdash e_1\ e_2 \longrightarrow v} \qquad \text{(APPCON)}$$

The $\delta(v_1, v_2)$ relation describes the application of the primitive operators and is defined by

$$
\begin{array}{rclcrcl}
\delta(+, i) & = & +_i & & \delta(-, i) & = & -i \\
\delta(\times, i) & = & \times_i & & \delta(/2, i) & = & i\ \mathit{div}\ 2 \\
\delta(+_i, i') & = & i + i' & & \delta(\mathit{pos?}, i) & = & \mathit{Clos}([], x, \lambda y.x),\ \mathit{if}\ i > 0 \\
\delta(\times_i, i') & = & i \times i' & & & & \mathit{Clos}([], x, \lambda y.y),\ \mathit{if}\ i \leq 0
\end{array}
$$

The closures resulting from the positive test are standard encodings for booleans.

I am currently assuming that the operational semantics used express evaluation until completion. Alternatively, semantics can express single-step evaluation (state transitions). While these two forms are equally descriptive, complete-evaluation semantics are frequently more convenient for describing high level languages. Single-step semantics can be cumbersome as they require the full specification of the evaluation order which is frequently an unnecessary level of detail. Contrast this

with (APP) above, which does not specify in what order the function and argument are evaluated. For machines, the step-step reduction form is traditional and usually more convenient. To provide a unified framework, I will present all operational semantics in the same manner, eliminating the artificial distinction between languages and machines.

Preliminary work indicates that it is simpler to obtain complexity information from the profiling semantics when they use environments rather than substitution. Furthermore, it seems simpler to translate an environment-based language model to a machine model (*cf.* Section 4) than it would be for a substitution-based model.

The profiling semantics extends the operational semantics to also describe the relevant costs c, and is defined by the relation $E \vdash e \longrightarrow v; c$. Here, the cost measure is intended to determine the serial time complexity of evaluating an expression. At the language-level, the cost measure that intuitively models time is the number of reductions in the evaluation derivation. Later, this intuition must be formalized. Thus, each reduction is of unit cost, and the costs of evaluating multiple subexpressions are summed. A common alternative is to count the number of "important" reductions. Since applications are the only reductions representing significant computation, it would be adequate to count only those.

The rules to define the profiling semantics are extensions of the operational rules:

$$\frac{E(x) = v}{E \vdash x \longrightarrow v; 1} \qquad (\mathrm{VAR_P})$$

$$E \vdash k \longrightarrow k; 1 \qquad (\mathrm{CONST_P})$$

$$E \vdash \lambda x.e \longrightarrow \mathit{Clos}(E, x, e); 1 \qquad (\mathrm{LAM_P})$$

$$\frac{E \vdash e_1 \longrightarrow \mathit{Clos}(E', x, e); c_1 \qquad E \vdash e_2 \longrightarrow v_2; c_2 \qquad E'[x \mapsto v_2] \vdash e \longrightarrow v; c_3}{E \vdash e_1\ e_2 \longrightarrow v; c_1 + c_2 + c_3 + 1} \qquad (\mathrm{APPLY_P})$$

$$\frac{E \vdash e_1 \longrightarrow v_1; c_1 \qquad E \vdash e_2 \longrightarrow v_2; c_2 \qquad \delta(v_1, v_2) = v}{E \vdash e_1\ e_2 \longrightarrow v; c_1 + c_2 + 1} \qquad (\mathrm{APPCON_P})$$

We are primarily interested in derivations with an empty environment, so that the expression encapsulates the entire program. For example,

$$[] \vdash (\lambda x.x)(\lambda y.y) \longrightarrow \mathit{Clos}([], y, y); 4$$

I.e., the identity function applied to itself (the renaming of variables is unimportant) returns the identity function expressed as a closure in four reduction steps.

The simplicity of this profiling semantics reflects the simplicity of the language and cost measure. The definition of a profiling semantics must be based on an operational semantics that captures all the information necessary to measure the costs. For example, cost measures based on run-time reference counts necessitate a two-level store model, rather than the simpler environment model, even if this is not necessary for the operational semantics.

This particular profiling semantics makes numerous assumptions for the purpose of simplification. For this example, we are interested only in asymptotic complexity, so it is adequate to state that all primitive applications require some constant amount of time represented by one reduction step. Similarly, the time for arithmetic operations is here considered independent of the size of the

integers, although "bignum" arithmetic is known to be expensive (*cf.* the logarithmic cost criterion of traditional complexity theory for RAMs). As is common, realistic memory considerations are also ignored.

Given a language and operational semantics, many profiling semantics are possible using different cost measures. Another measure commonly used in traditional complexity analysis is the space needed for evaluation. Other possible measures, not all relevant for this language, include the maximum size of a data structure, the total number of memory allocations made, instruction counts, and the maximum number of useful processors.

As part of the dissertation, I plan to provide examples of profiling semantics using some of the mentioned language and semantics features. These include

- higher-order functions,
- other costs measures such as space,
- tail-recursion,
- vectors and recursive datatypes,
- multiple syntactic classes including declarations,
- and possibly side-effects, continuations, *etc.*

Further examples are described in Section 5. I do not intend to examine other programming language paradigms such as logic programming or message passing.

3 Obtaining Algorithm Complexities

To obtain the costs for evaluating an expression, we "simply" ignore the values in the profiling semantics. Ignoring the costs in the profiling semantics *is* simple and results in the original operational semantics. But of course, not all values involved in evaluation can be ignored, as here the cost c_3 in (APPLY_P) depends on the values of the function and argument.

But a profiling semantics only defines the cost of evaluating an expression. *E.g.*, the cost of evaluating *fact* 1 is 55, where we use the definitions

$$\begin{aligned} \mathit{fact} &\equiv Y(\lambda \mathit{fact}'.\lambda n.(\mathit{pos?}\ n)\ (\lambda d.\times\ n\ (\mathit{fact}'(+\ n\ -1)))\ (\lambda d.1)\ 0 \\ Y &\equiv \lambda g.(\lambda f.g\ (\lambda a.f\ f\ a))\ (\lambda f.g\ (\lambda a.f\ f\ a)) \end{aligned}$$

The expression abbreviated by *fact* is the factorial function (except that negative integers are mapped to 1). The expression Y is known as the Y-combinator and is a higher-order function used in this sparse language to define recursive functions. The large number of reductions involved in this example is largely due to the use of Y.

More typically, we are interested in the complexity of an algorithm (a function) which expects input (an argument). *E.g.*, for any i, the cost of evaluating *fact* i is $31i + 24$, if $i \geq 0$, and 24, if $i < 0$.[1] To prove this, we must examine the derivation of the costs for *fact* i and extract a recurrence equation on these costs.

[1] By definition of *pos?*, *fact* applied to a non-integer does not have a successful evaluation.

For the derivation, we must evaluate the function, the argument, and their application, and use (APPLY$_P$). For the function, we have $[] \vdash \mathit{fact} \longrightarrow \mathit{Clos}([\mathit{fact'} \mapsto v_{\mathit{fact'}}], n, \mathit{body}); 10$, where

$$\begin{aligned} v_g &\equiv \mathit{Clos}([], \mathit{fact'}, \lambda n.\mathit{body}) \\ v_f &\equiv \mathit{Clos}([g \mapsto v_g], f, g\ (\lambda a.f\ f\ a)) \\ v_{\mathit{fact'}} &\equiv \mathit{Clos}([g \mapsto v_g, f \mapsto v_f], a, f\ f\ a) \\ \mathit{body} &\equiv (\mathit{pos?}\ n)\ (\lambda d.\times\ n\ (\mathit{fact'}\ (+\ n\ -1)))\ (\lambda d.1)\ 0) \end{aligned}$$

Since we are assuming that the argument is given as the constant i, we have $[] \vdash i \longrightarrow i; 1$.

Now we examine the costs of the application. For the base case, when $i \leq 0$,

$$[\mathit{fact'} \mapsto v_{\mathit{fact'}}, n \mapsto i] \vdash \mathit{body} \longrightarrow i; 12$$

When $i > 0$,

$$[\mathit{fact'} \mapsto v_{\mathit{fact'}}, n \mapsto i] \vdash \mathit{body} \longrightarrow i; 31 + c$$

where $[\mathit{fact'} \mapsto v_{\mathit{fact'}}, n \mapsto i - 1 \vdash \mathit{body} \longrightarrow i; c$. Simply abbreviating the relevant parts of these derivations as equations gives

$$\begin{aligned} f'(i) &= 31 + f'(i-1) \quad \mathit{if}\ i > 0 \\ f'(i) &= 12 \qquad\qquad\ \ \mathit{if}\ i \leq 0 \\ f(i) &= 12 + f'(i) \end{aligned}$$

The number 12 in the last equation is the cost of evaluating the function and argument plus the overhead of 1 for (APPLY$_P$). Solving for $f(i)$ gives the costs above for evaluating *fact* i, and thus *fact* has the expected $O(i)$ complexity.

Similar cost semantics in automatic complexity analysis have simplified the extraction of recurrences by using *cost functions* in the semantics. Essentially, these cost functions add information to the semantics to label those recurrences that may be needed. Unnamed recursive functions, such as those defined with Y, are not successfully analyzed by that method. To provide cost functions that are suitable for automatic solvers, languages have been restricted to having only named, top-level functions.[2]

However, cost functions complicate the semantics by adding information which is unnecessary for the definition of costs. I prefer to keep the profiling semantics as simple as possible, since this is what must be provided by the user. An automatic complexity analyzer can then use whatever tools necessary to perform the analysis, including such a partial evaluation of *fact* to discover cost functions.

The unrolling of the evaluation proof tree for *fact* i is an instance of a general technique which effectively finds the appropriate cost functions. As long as substitution is not used in the semantics[3], all branches of such proof trees either terminate or repeat the expression being evaluated (usually with different contexts). Recurrence equations can be formed from these two cases, with the former as base cases. Side conditions such as $i > 0$ arise from conditional branches.

---

[2]With currying, this does not restrict the expressiveness of the language [26].

[3]This is possible for at least call-by-value, call-by-name, and call-by-need languages without side-effects or control abstractions.

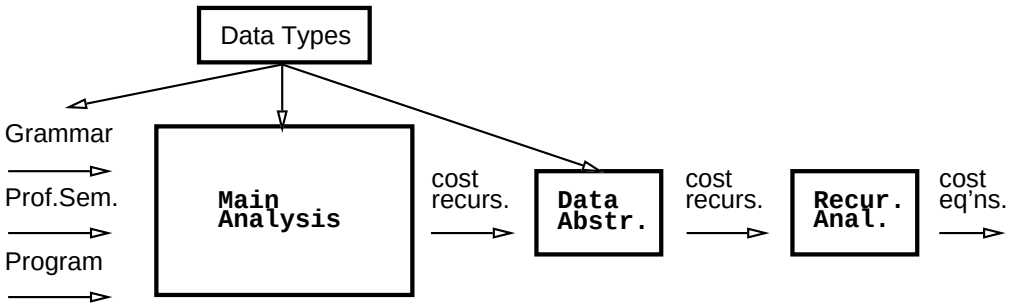

Figure 1: Structure of an automatic complexity analyzer

In general, however, the cost of evaluating a function applied to any argument cannot be expressed consisely for all possible arguments. Frequently, a consise description can be made relative to some abstraction of the possible arguments. Consider extending the λ-calculus with the list primitives *nil*, *nil?*, *hd*, *tl*, and *cons*. The length and reverse functions can be defined to have the expected asymptotic complexity of $O(n)$ where n is the length abstraction of the argument list.

Functions defined by structural induction have complexities that are described relative to structurally inductive abstractions such as the length of a list, depth of a tree, or height of a tree. Automatic complexity analyzers have primarily been focused on such functions. These abstractions, however, can be arbitrarily complex and are generally based on the representation of data objects in the program.

A block diagram of the analyzer's structure is given in Figure 1. The heart of the main analysis will be the unrolling of the evaluation proof tree. Following previous analyzers [17, 18, 34], I expect to do data abstraction on the recurrence equations generated by the main analysis. I will not perform recurrence analysis to generate closed forms from the recurrence equations, unless I can use pre-existing code. As indicated, information about the allowable data types must be known to all stages of the analysis.

4 Relating language models to machine models

Language-based complexity results can be useful by themselves. For example, consider the given λ-calculus without the arithmetic constants. It is well-known that the best predecessor function on the Church numeral encoding of natural numbers requires $O(n)$ reductions to compute the predecessor of the encoding of n. Thus, this standard encoding is not efficient with respect to reductions. Similarly, it can be shown that other encodings are more efficient with respect to reductions (but less efficient with respect to the size of expressions and values) [20]. Informally, the language model with arithmetic constants is more cost-expressive that the one without. Such comparisons can be made entirely at the language level.

However, such results do not show whether, for example, the Church numeral predecessor can be implemented in $O(n)$ *time*. For this, the language model must be related to a machine model in which time costs are measured. Then it must be shown that there is a translation such that the desired relation between the number of reductions and time holds. This translation is in essence a compiler.

By compiling a language model to a machine model, I am assuming that the machine model

embodies a more basic model of computation with more realistic complexity behavior. Despite their unrealistic assumptions, *e.g.*, of infinite memory and arbitrary precision integers, I will consider the RAM and PRAM as the basic machine models. Others have shown the relation of the PRAM to architecture-dependent models of parallelism [2, 15, 16, 21, 31, 32].

As part of the translation to the machine model, it is often convenient to use an intermediate step to help decompose the translation into separate components. What intermediate language is most convenient depends on both the language and the costs being measured. For the example profiling semantics, a suitable intermediate is the SECD machine [14].

A state of the SECD machine consists of a stack S of values, an environment E, a control list C of expressions, and a dump list D of (S, E, C) triples. The operational semantics are then given the non-traditional form of $S; E; C; D \longrightarrow v; c$, as defined by the following rules.

$$[v]; E; \mathit{nil}; \mathit{nil} \longrightarrow v; 1 \qquad (\text{S-DONE}_\text{P})$$

$$\frac{v :: S'; E'; C'; D \longrightarrow v; c}{v :: S; E; \mathit{nil}; (S', E', C') :: D \longrightarrow v; c + 1} \qquad (\text{S-NIL}_\text{P})$$

$$\frac{E(x) :: S; E; C; D \longrightarrow v; c}{S; E; x :: C; D \longrightarrow v; c + 1} \qquad (\text{S-VAR}_\text{P})$$

$$\frac{k :: S; E; C; D \longrightarrow v; c}{S; E; k :: C; D \longrightarrow v; c + 1} \qquad (\text{S-CONST}_\text{P})$$

$$\frac{\mathit{Clos}(E, x, e) :: S; E; C; D \longrightarrow v; c}{S; E; (\lambda x.e) :: C; D \longrightarrow v; c + 1} \qquad (\text{S-LAM}_\text{P})$$

$$\frac{S; E; e_2 :: e_1 :: \mathbf{app} :: C; D \longrightarrow v; c}{S; E; (e_1\ e_2) :: C; D \longrightarrow v; c + 1} \qquad (\text{S-APPLY}_\text{P})$$

$$\frac{\mathit{nil}; E'[x \mapsto v]; [e]; (S, E, C) :: D \longrightarrow v; c}{\mathit{Clos}(E', x, e) :: v :: S; E; \mathbf{app} :: C; D \longrightarrow v; c + 1} \qquad (\text{S-APP}_\text{P})$$

$$\frac{\delta(v_1, v_2) :: S; E; C; D \longrightarrow v; c}{v_1 :: v_2 :: S; E; \mathbf{app} :: C; D \longrightarrow v; c + 1} \qquad (\text{S-APPCON}_\text{P})$$

We are primarily interested in derivations with an empty stack, environment, and dump, and with a singleton control list. These correspond to the normal start states for the SECD machine. Using the same expression as for the profiling semantics, we have

$$\mathit{nil}; []; [(\lambda x.x)(\lambda y.y)]; \mathit{nil} \longrightarrow \mathit{Clos}([x \mapsto \mathit{Clos}([], y, y)], y, y); 7$$

The reduction cost in the SECD model is higher than that in the λ-calculus model. It is easy to show that the SECD model's cost is bounded by three times the λ-calculus model's cost, plus one. The factor of three arises by the correspondence of one (APPLY_P) reduction to one each of (S-APPLY_P), (S-APP_P), and (S-NIL_P). Similarly, one (APPCON_P) corresponds to one each of (S-APPLY_P) and (S-APPCON_P). The extra cost of one is the final (S-DONE_P) reduction. This suggests an alternate SECD model, with costs of 0 in (S-DONE_P) and c in the conclusions of (S-APPLY_P), (S-APP_P), and (S-NIL_P). Then, the costs would correspond exactly with that of the λ-calculus model. For an asymptotic analysis, these constant factors are irrelevant.

To complete the translation of the λ-calculus to the machine model, we must map the SECD onto the RAM. The details are beyond the scope of this proposal, but I will outline the main ideas. The main issue is mapping each of the data structures used in the SECD machine. Mapping tuples is straightforward. Lists and stacks can be implemented with a heap, and garbage-collection is unnecessary since the RAM assumes infinite memory. Environments can be implemented with a balanced tree, with variable names as the keys.

The primitive tuple, list, and stack operations only require constant time in the RAM. Environment access requires time proportional to the logarithm of the size of the environment. Extending the environment can be implemented in the same logarithmic complexity; however, updating the tree must be non-destructively, sharing pointers to the unchanged parts of the tree. The size of the environment is obviously bounded by the number of variables in the program. By renaming variables to reuse the same name as much as possible, the number of variables can be reduced to the maximum deBruijn index ($dB(\cdot)$) of the original program. Also, the variable names can be represented by numbers so that comparisons on them, necessary for environment lookups, require constant time. So, we have the following theorem.

Theorem 1 *The λ-calculus can be implemented such that if a function has a reduction complexity of $O(f(n))$ in the above model, then it has a time complexity of $O(f(n) \log dB(f))$ in the RAM model.*

Some compiler optimizations improve the complexity of a program. For example, a "tail recursive" function can be implemented to use only constant stack space, rather than an amount proportional to the number of function calls. By definition of Scheme [7], any implementation of Scheme must have this optimization. On the other hand, implementations of other languages frequently do not implement this optimization. Instead of viewing this as a compiler changing the algorithmic complexity of Scheme, I argue that the profiling semantics must reflect any such requirements of a language definition. Another example of this is found in Section 5.

5 Comparing the Cost-Expressiveness of Language Models

The second half of the dissertation will provide two extended examples of working with language complexity models. The purpose is to demonstrate the usefulness of the language models of complexity. This will center on comparing the cost-expressiveness of models of various parallel languages.

I will work with two languages which give distinct models of parallelism. Because of the emphasis on modern high-level languages, both of these will use higher-order functions. But for simplicity, they will not have side-effects or non-determinism. Also for simplicity, these languages will be very basic and not incorporate unnecessary constructs. Both are also *sequential*, *i.e.*, their evaluation could be serialized while preserving meaning.

For each, I will use the cost measures of *work* and *steps*, informally known as the work-time paradigm, which has been used to describe parallel algorithm complexity at an abstract level [11, 12, 29]. The work is the number of serial reductions. The step cost describes the number of (abstract) time units needed for evaluation, assuming an infinite number of processors and unit communication costs. Brent [6] showed how to relate the work and step complexities of an algorithm to a machine with a given finite number of processors.

I will relate the language-based complexity definitions to definitions based on the PRAM. This will include relating results on work-efficiency at the two levels.

First, I will compare the cost-expressiveness of the two languages to each other. Next, I will show how the cost-expressiveness of the data-parallel language depends on the primitives used in the language.

Parallel λ-calculus

The first language allows only a bounded number of processes to be created in a finite time. It is the call-by-value λ-calculus such that the subexpression of an application are evaluated in parallel. It has long been argued informally that this could serve as a simple model for a parallel language. The language has a small set of arithmetic constants.

The most relevant rules of the profiling semantics for this language are

$$\frac{E \vdash e_1 \longrightarrow \mathit{Clos}(E', x, e); w_1, s_1 \qquad E \vdash e_2 \longrightarrow v_2; w_2, s_2 \qquad E'[x \mapsto v_2] \vdash e \longrightarrow v; w_3, s_3}{E \vdash e_1\ e_2 \longrightarrow v; w_1 + w_2 + w_3 + 1, \max(s_1, s_2) + s_3 + 1} \quad (\text{P-APPLY}_\text{P})$$

This defines the step cost of an application to involve the maximum of the step costs of its subexpressions. This effectively states that these subexpressions must be evaluated in parallel.

In order to relate this language model to the PRAM model, a parallel generalization of the SECD machine will be used as an intermediate model. The costs in the parallel SECD model will be within a constant factor of those in the language model. The costs of the PRAM will be within a constant factor times the logarithm of the number of maximum deBruijn variable index of an expression. Because of the use of a microanalysis, detailed results will be obtained on precisely what these constants are.

This λ-calculus is too sparse to be a convenient programming language, so I will introduce a model with ML-like syntax and recursive datatypes. Programs in this language, which consist of a sequence of datatype and function declarations and an expression to be evaluated, can be translated to the parallel λ-calculus model. This translation uses a tagging implementation of products and sums (and thus, datatype constructors) much like a realistic compiler and introduces only constant work and step overheads.

A Data-Parallel Language

The other language is based on the ability to create an unbounded number of processes in finite time. It incorporates random access, in the form of arrays, and a construct to map a f function across an array $[a_1, \ldots, a_n]$, creating a process for each application $f\ a_i$. Conceptually, the array elements are stored on n separate processors, so that array manipulations are disguised forms of communication. This is the data-parallel paradigm, which includes languages such as *Lisp, Parallation Lisp, Fortran90, HPF, and NESL.

Among these languages, there is disagreement on what basic array (communication) primitives should be used. Such primitives include reads, writes, scans (all prefix sums), permutes, and filtering. Part of the thesis will investigate how the choice of such primitives effects the cost-expressiveness of the language model.

The core of the language will be be the following expressions, similar to those of NESL.

$C \in \mathit{Constructors}$

$$c \in \mathit{Constants}$$
$$e \in \mathit{Expressions} \quad ::= \quad c \mid C\ e \mid [e_1, \ldots, e_n] \mid \lambda x.e \mid e_1 \mid e_2 \mid \{e_1 : x\ \mathit{in}\ e_2\} \mid \ldots$$

Constructors are defined by datatype declarations.

Two different profiling semantics will be used for this language, differing in the expected implementation of arrays. The first semantics will specify the naive copying semantics of functional arrays. Whenever an array is updated with new elements, the whole array must be copied, even if the original array is no longer needed.

This language is more cost-expressive than the parallel λ-calculus, given the same arithmetic primitives. Informally, this follows from the following two facts. The parallel λ-calculus can be modelled with constant overhead using parallelism on arrays of length two. And being able to create an unbounded number of processes each step does increase cost-expressiveness.

Giving the costs as a pair w, s of work and steps, two of the key rules of this profiling semantics are

$$\frac{D; E \vdash e_1 \longrightarrow \mathit{Clos}(E', x, e); w_1, s_1 \qquad D; E \vdash e_2 \longrightarrow v_2; w_2, s_2 \qquad D; E'[x \mapsto v_2] \vdash e \longrightarrow v; w_3, s_3}{D; E \vdash e_1\ e_2 \longrightarrow v; w_1 + w_2 + w_3 + 1, s_1 + s_2 + s_3 + 1} \quad (\text{D-APPLY}_\text{P})$$

$$\frac{D; E[x \mapsto v'_i] \vdash e_1 \longrightarrow v_i; w_i, s_i \qquad \forall i \in \{1, \ldots, n\} \qquad D; E \vdash e_2 \longrightarrow [v'_1, \ldots, v'_n]; w', s'}{D; E \vdash \{e_1 : x\ \mathit{in}\ e_2\} \longrightarrow [v_1, \ldots, v_n]; w' + 1 + \sum_{1 \le i \le n} w_i, s' + 1 + \max_{1 \le i \le n} s_i} \quad (\text{D-EACH}_\text{P})$$

where D contains information about datatype declarations. The application rule is like the serial rule (APPLY$_\text{P}$), since parallelism is not associated with application. In the other rule, the body e_1 is evaluated for each of the elements v'_i, with x bound in each evaluation to a different element. These n evaluations of e_1 are to be done in parallel, as the specified step cost involves the maximum of the steps of these evaluations.

The second semantics will specify a destructive updating semantics, using reference counting to detect when a single reference exists to an array. Ignoring costs, these two language implementations could share the same semantics, since they are extensionally equivalent. However, this operational semantics does not have information about reference counts which is necessary to define the appropriate profiling semantics. In order to define reference counting, the operational semantics must fully define evaluation order. Not surprisingly, the resulting profiling semantics gives a more cost-expressive model than that with copying arrays.

Possible other languages

Time allowing, I may investigate other functional models of parallelism. One such model has been called a *lenient* language [23]. To evaluate an application, the subexpressions are evaluated in parallel. Unlike the parallel λ-calculus, however, evaluation of the body of the function begins as soon as it is known, without waiting for the argument evaluation to complete. This is similar to call-by-need, but may result in more work. The other model is based on the parallel-or primitive. When applied, this primitive evaluates its two arguments in parallel. It returns *true* as soon as either argument returns *true*, not waiting for the other to complete. Thus, the step cost of *por* e_1 e_2 is the minimum of the step costs of e_1 and e_2.

6 Related work

As has been emphasized, this is an extension of traditional complexity analysis, *e.g.*, [1]. Usually complexities are defined relative to a machine-oriented model which more-or-less corresponds to an idealized machine architecture. These models are based on a state transition function, rather than a complete-evaluation relation. Also, I am formally incorporating the costs into the semantics of the language.

A primary use of language-based cost models has been in automatic complexity analysis. Given a program, such a system attempts to automatically derive complexity bounds for this program. This analysis is undecidable, by the halting problem. Systems have been developed primarily for first-order serial call-by-value functional languages with recursive datatypes [17, 18, 26, 34, 35, 36], although one has used parallelism with arrays [37, 38, 39]. During this analysis, the program is translated into one which computes the cost of executing the original program. Later systems defined these translations via cost models based on operational semantics, although earlier systems simply argued this step informally.

Zimmermann [37] provided the only automatic complexity analyzer with any language flexibility. A facility was provided to allow the user to specify necessary information about the datatypes of a language. The other aspects of the language and semantics were fixed.

When defining the costs of lazy, or call-by-need, languages, the difficulty is in keeping track how much of each expression will need to be evaluated [4, 25, 26, 27, 28, 33]. Sands seperates the evaluation relation into two — one to head-normal form and one to a value. Bjerner and Wadler both add information to denotational semantics about what must be evaluated. Also, neither attempts to obtain complexities from the derived costs. Because the definitions are more complicated, automatic analysis for these languages has not been attempted.

Cost information can also be added to denotational semantics [4, 10, 23, 24, 30, 37, 38, 39]. At least in the existing work, this approach leads to more complicated definitions of models. More generally, an *intensional* semantics is the extension of a denotational semantics to incorporate information about *how* results are computed, rather than just *what* is computed. Not all of the above attempt to obtain complexities from the costs. Gurr's elegant use of monoids to structure the costs in such semantics proved too limiting for some models, such as needed for parallel languages.

Cost information can be treated as a computational effect and added to type systems. So far, cost systems as effects have been limited to obtaining useful information only for first-order languages. Because effects systems do not solve arithmetic recurrences, these systems either restrict the language to avoid recurrences [22] or simply return the maximal cost effect when recurrences would be needed [8]. Soundness proofs should be extended to relate the static cost information to the dynamic-oriented cost information obtained through profiling semantics.

Flexible semantics frameworks have been developed for specifying monitors [13]. Profilers can be defined in these frameworks which count the costs of evaluating a program. The form of these semantics are substantially different. The profiling information is added using "hooks" provided by the underlying semantics, making a compositional definition of costs awkward. Furthermore, in these monitors, there is no attempt to derive complexities from these costs.

Previous definitions of expressiveness have been based on the definability of programming constructs within the language [9, 19]. My approach is instead based on the lower bounds of the costs of the definable algorithms in a language. These approaches are not directly comparable.

7 Completed Work, Goals, and Contributions

Research is expected to concentrate on two major points. First, language models of complexity can be defined by profiling semantics. Further research will concentrate on systematically obtaining complexity information from profiling semantics; determining restrictions of the form of the semantics that are necessary or helpful for this; and relating language models of complexity to machine models.

To demonstrate the generality of the approach, I will implement a prototype of an automatic complexity analyzer that is parameterized by a profiling semantics. This should also give me concrete feedback about the strengths and weakness of my theoretical work. However, rather than sophisticated analyses specialized for small classes of languages, my emphasis will be on analyses of wide applicability and how to generalize the special cases. Therefore, it is likely that my implementation will not be as sophisticated as the previous single-language analyzers.

Second, the cost-expressiveness of some language models will be compared, giving one measure of their appropriateness as the basis of realistic programming languages. This will be a non-trivial use of the tools developed in the first half of the thesis. One comparison will be between at least the two parallel languages described in Section 5. I will also examine how the cost-expressiveness of the data-parallel language depends on the primitives included in the language.

Thus, the primary expected contributions of this work are

- a simple method (profiling semantics) for defining complexity measures at a language-level, which can then be related to traditional machine models;
- general techniques to derive cost functions and complexity measures from a wide variety of profiling semantics, suitable for automatic complexity analysis;
- a prototype implementation of a automatic complexity analyzer generalized for different languages and semantics;
- definitions of a number of different interesting cost models;
- a definition of expressiveness based on cost information; and
- comparisons of the cost-expressiveness of some parallel language models.

The last point may include complexity results about specific basic algorithms such as permuting and sorting to highlight the usefulness of profiling semantics.

So far, I have experimented with various profiling semantics and begun implementation of the prototype analyzer. Sketches of a profiling semantics for a wide variety of languages support my claims that profiling semantics are a useful definitional tool. Early implementation work supports my claims that some analysis techniques are sufficiently general.

Some aspects of analysis may be problematic, however. It is not yet clear whether more datatype-specific analyses such as vector length analysis or detection of techniques such as pointer doubling will generalize, or if numerous special cases must be made. Side-effects complicate any analysis, which is why I have decided to emphasize languages without side-effects, at least initially. The remainder of the thesis does not depend on the generality of analysis, but only on the generality of the definitional aspects of profiling semantics.

It is *not* a central goal to provide accurate estimates of execution time, and similarly detailed cost measures. Language models for such a cost measure would have to include most aspects of

the compiler and the hardware. Creating a profiling semantics and performing any analysis would be incomprehensible and would need to be highly automated. When such specialized results are desired, standard profiling tools may be more practical, although they do not provide the same general complexity information.

8 Time schedule

Activity	Time (months)
Determining the form of, providing examples of, and deriving complexities from profiling semantics	3.5
Implementing prototype analyzer	4.5
Cost-expressiveness of parallel language models	4.5
Cost-expressiveness of data-parallel operators	1.5
Writing dissertation	4
Total	18

Adding a month for sanity checks (vacations), my expected completion date is November 1995.

References

[1] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[2] H. Alt, T. Hagerup, K. Kehlhorn, and F. P. Preparata. Deterministic simulation of idealized parallel computers on more realistic ones. *SIAM Journal on Computing*, 16:808–835, 1987.

[3] Amir M. Ben-Amram and Zvi Galil. On pointers versus addresses. *Journal of the ACM*, 39(3):617–648, July 1992.

[4] Bror Bjerner and Sören Holmstroöm. A compositional approach to time analysis of first order lazy functional programs. In *4th International Conference on Functional Programming Languages and Computer Architecture*. Springer-Verlag, September 1989.

[5] Guy E. Blelloch. NESL: A nested data-parallel language (version 2.6). Technical Report CMU-CS-93-129, Carnegie Mellon University, April 1993.

[6] R. P. Brent. The parallel evaluation of general arithmetic expressions. *Journal of the ACM*, 21(2):201–206, 1974.

[7] William Clinger and Jonathan Rees. Revised^4 report on the algorithmic language Scheme. *LISP Pointers*, IV(3):1–55, July–September 1991.

[8] Vincent Dornic, Pierre Jouvelot, and David K. Gifford. Polymorphic time systems for estimating program complexity. In *JTASPEFL*, October 1991.

[9] Matthias Felleisen. On the expressive power of programming languages. *Science of Computer Programming*, 17(1–3):35–75, December 1991.

[10] Douglas J. Gurr. *Semantic Frameworks for Complexity.* PhD thesis, University of Edinburgh, January 1991.

[11] Joseph JáJá. *An Introduction to Parallel Algorithms.* Addison-Wesley, Reading, Mass., 1992.

[12] R. M. Karp and V. Ramachandran. Parallel algorithms for shared memory machines. In J. Van Leeuwen, editor, *Handbook of Theoretical Computer Science — Volume A: Algorithms and Complexity.* MIT Press, Cambridge, Mass., 1990.

[13] Amir Kishon, Paul Hudak, and Charles Consel. Monitoring semantics: A formal framework for specifying, implementing, and reasoning about execution monitors. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 338–352, June 1991.

[14] P. J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6:308–320, 1964.

[15] F. Luccio, A. Pietracaprina, and G. Pucci. A new scheme for the deterministic simulations of prams in vlsi. *Algorithmica*, 5:529–544, 1990.

[16] Kurt Mehlhorn and Uzi Vishkin. Randomized and deterministic simulations of prams by parallel machines with restricted granularity of parallel memory. *Acta Informatica*, 21:339–374, 1984.

[17] D. Le Métayer. Mechanical analysis of program complexity. In *SIGPLAN Symposium on Language Issues in Programming Environments*, 1985.

[18] Daniel Le Métayer. Ace: An automatic complexity evaluator. *ACM Topics on Programming Languages and Systems*, 10(2):248–266, April 1988.

[19] John C. Mitchell. On abstraction and the expressive power of programming languages. In *Theoretical Aspects of Computer Software*, pages 290–310, September 1991.

[20] Michel Parigot. Programming with proofs: A second order type theory. In H. Ganzinger, editor, *2nd European Symposium on Programming*, volume 300 of *Lecture Notes in Computer Science*, pages 145–159. Springer-Verlag, 1988.

[21] A. G. Ranade. How to emulate shared memory. *Journal of Computer and System Sciences*, 42(3):307–326, June 1991.

[22] Brian Reistad and David K. Gifford. Static dependent costs for estimating execution time. July 1993.

[23] Paul Roe. Calculating lenient programs' performance. In Simon L Peyton Jones, Graham Hutton, and Carsten Kehler Holst, editors, *Functional Programming, Glasgow 1990*, Workshops in computing. Springer-Verlag, August 1990.

[24] Mads Rosendahl. Automatic complexity analysis. In *4th International Conference on Functional Programming Languages and Computer Architecture.* Springer-Verlag, 1989.

[25] David Sands. Complexity analysis for a lazy higher-order language. In *Glasgow Functional Progamming Workshop*, Springer-Verlag Workshops in Computing Series. Springer-Verlag, 1989.

[26] David Sands. *Calculi for Time Analysis of Functional Programs.* PhD thesis, University of London, Imperial College, September 1990.

[27] David Sands. Operational theories of improvement in functional languages (extended abstract). 1991.

[28] David Sands. Time analysis, cost equivalence and program refinement. In *11th Conference on Foundations of Software Technology and Theoretical Computer Science*, Lecture Notes in Computer Science. Springer-Verlag, December 1991.

[29] Y. Shiloach and U. Vishkin. An $O(n^2 \log n)$ parallel Max-Flow algorithm. *Journal of Algorithms*, 3:128–146, 1982.

[30] Jon Shultis. On the complexity of higher-order programs. Technical Report CU-CS-288-85, University of Colorado, Boulder, January 1985.

[31] E. Upfal and A. Widgerson. How to share memory in a distributed system. *Journal of the ACM*, 34:116–127, 1987.

[32] L. G. Valiant and G. J. Brebner. Universal schemes for parallel communication. In *ACM Symposium on Theory of Computing*, pages 263–277, 1981.

[33] Philip Wadler. Strictness analysis aids time analysis. In *15th ACM Symposium on Principles of Programming Languages*, January 1988.

[34] Ben Wegbreit. Mechanical program analysis. *Communications of the ACM*, 18(9), September 1975.

[35] Paul Zimmermann and Wolf Zimmermann. The automatic complexity analysis of divide-and-conquer algorithms. Technical Report 1149, Institut National de Récherche en Informatique et en Automatique, Rocquencourt, December 1989.

[36] Paul Zimmermann and Wolf Zimmermann. The automatic complexity analysis of divide-and-conquer algorithms. *Computer and Information Sciences VI*, 1991.

[37] Wolf Zimmermann. Automatic worst case complexity analysis of parallel programs. Technical Report TR-90-066, International Computer Science Institute, December 1990.

[38] Wolf Zimmermann. The automatic worst case analysis of parallel programs: Simple parallel sorting and algorithms on graphs. Technical Report TR-91-045, International Computer Science Institute, August 1991.

[39] Wolf Zimmermann. Complexity issues in the design of functional languages with explicit parallelism. In *International Conference on Computer Languages*, pages 34–43, 1992.