ON THE COMPLEXITY OF HIGHER-ORDER PROGRAMS

by

Jon Shultis*

CU-CS-288-85 January, 1985

*Department of Computer Science, University of Colorado, Boulder, Colorado 80309

On the Complexity of Higher-Order Programs

Jon Shultis
Department of Computer Science
University of Colorado
Boulder, CO 80309

Abstract

A logic for analyzing the time complexity of programs using higher-order functions is presented. The subject of the study is a simple functional language with static binding and call-by-value as its only parameter discipline. The language is defined by a denotational semantics that models both the functionality and cost of programs. The logic was derived from the semantic model, and tested against an implementation of the language based directly on the semantics. The use of the logic is demonstrated for a number of examples including data structure comparison and higher-order recursive functions.

On the Complexity of Higher-Order Programs

Jon Shultis
Department of Computer Science
University of Colorado
Boulder, CO 80309

1. Introduction

Traditional complexity theory is inadequate for reasoning about programs that use higher-order functions, because it explains how to determine the cost of computing a value, but not how to determine the cost of subsequently applying that value if it is a function. For example, the cost of *map f lis* depends not only on the value and cost of *f* and *lis*, but also on how much it costs to apply *f* to an element of *lis*. All three aspects of *f* - its value, its cost, and its cost of use - must be considered in describing the complexity of *map*.

Our purpose here is to extend complexity theory to include the analysis of programs using higher-order functions. The subject of our theory is a simple functional language consisting of a few primitives, conditional expressions, λ-abstraction, and recursion.

Our approach is formal, and proceeds from a detailed semantic model to an axiomatic theory of complexity in which analyses are performed. The formal development is appropriate because our intuition about the complexity of higher-order programs is so weak. It is therefore important to see how the methods arise from the underlying mathematics. Once one grasps the principles in the formal setting, one can adopt a more casual style with confidence.

The scope of the current study is limited to the analysis of time complexity for a language with static binding and call-by-value as its only parameter discipline. We assume a fairly conventional implementation based on a heap of activation records and closures for a single-processor von Neumann machine. Storage is assumed to be unlimited, with constant access time, and there is no automatic optimization of the code, and no garbage collection. Extensions of our theory to a more realistic setting are discussed briefly at the end of the paper.

2. Cost Model for a Simple Functional Programming Language

Every logical theory is associated with a model or class of models. Which comes first - the theory or the model - depends on the goals of the research. To understand how to reason about something, one must start with the thing and develop a theory about it. Accordingly, our study begins with the specification of a model for a simple functional programming language, using the denotational method [5].

For the theory to be about complexity, the cost of running a program must be included in the model, either directly or indirectly. We call such a model a *cost model*, to emphasize its modelling of costs in addition to the more common modelling of functionality or abstractions thereof, as in data flow analysis and type checking.

The model is presented in the usual format (see, e.g., [3]). A few points about notational conventions are in order, however. First, we use the double brackets $[\![$ and $]\!]$ to denote the semantic function. It is more usual to use them for "syntax brackets", adding more symbols for the semantic function, but this is needless notational clutter.

The syntax of the programming language resembles that of the metalanguage in several respects; both use λ, tupling, and the form $b \rightarrow x, y$ for conditional expressions. It is always possible to determine which is meant from context, but readers who do not regularly read denotational specifications should keep the distinction clearly in mind when reading the equations. To aid in distinguishing between them, angle brackets $< >$ have been used to delimit tuples in the metalanguage, whereas parentheses are used for that purpose in the subject language.

One notation is used both for functional updating and substitution, viz. $a[b/c]$. When a is an expression, free substitution of b for c is indicated. When a is a function, updating a to have value b at point c is indicated. Many authors prefer to use distinct notations for these two concepts.

The last point to keep in mind is that we have omitted all injections and projections into and out of sum domains, and all isomorphisms in the interest of making the equations less cluttered. Thus, for example, it is left to the reader to infer that $\lambda <v_1,v_2>.<v_1+v_2,1>$ in the equation defining the primitive + has to be injected into the right summand of $\mathbf{Z}$ + [Val* $\rightarrow$ D], and an isomorphism applied to embed it in the domain Val.

Syntax:

$$\text{Exp} ::= \text{Num} \mid \text{Prim} \mid \text{Var} \mid e_0(e_1, \cdots, e_n)$$
$$\mid \lambda(x_1, \cdots, x_n).e \mid e_1 \rightarrow e_2, e_3 \mid rec\ x.e$$

$$\text{Prim} ::= \text{T} \mid \text{F} \mid + \mid - \mid * \mid \wedge \mid \vee \mid \neg \mid \leq$$

Semantic Domains:

$$\text{Cost} = \mathbf{N}$$

$$\text{Val} = \mathbf{Z} + [\text{Val}^* \rightarrow \text{D}]$$

$$\text{D} = \text{Val} \times \text{Cost}$$

$$\text{Env} = \text{Var} \rightarrow \text{Val}$$

Auxiliary Functions:

valof:D $\rightarrow$ Val

costof:D $\rightarrow$ Cost

Semantic Function:

$[\![\ ]\!]$: Exp $\rightarrow$ [Env $\rightarrow$ D]

Semantic Clauses:

$[\![\mathrm{n}]\!] \rho = <\mathrm{n},1>$

$[\![\mathrm{T}]\!] \rho = <1,1>$

$[\![\mathrm{F}]\!] \rho = <0,1>$

$[\![+]\!] \rho = <\lambda <v_1,v_2>. <v_1+v_2,1>, 1>$

similarly for -, *

$[\![\wedge]\!] \rho = <(\lambda <v_1,v_2>. v_1 = v_2 = 1 \rightarrow <1,1>, <0,1>), 1>$

similarly for $\vee$, $\neg$, $\leq$

$[\![x]\!] \rho = <\rho\ x, 1>$

$[\![e_0(e_1, \cdots, e_n)]\!] \rho =$

$(\lambda <rator,crator>.$

$(\lambda <res,cres>. <res,\ crator + cres + \sum_{i=1}^{n} costof([\![e_i]\!] \rho)>)$

$(rator <valof([\![e_1]\!] \rho), \cdots, valof([\![e_n]\!] \rho)>))([\![e_0]\!] \rho)$

$$[\![\lambda(x_1, \cdots, x_n).e]\!] \rho =$$

$$<\lambda <v_1, \cdots, v_n>. [\![e]\!] \rho[v_1/x_1, \cdots, v_n/x_n], \mathrm{n}>$$

$$[\![e_1 \rightarrow e_2, e_3]\!] \rho =$$

$$(\lambda <v,c>. (\lambda <v',c'>. <v', c+c'+1>)$$

$$(v=1\rightarrow [\![e_2]\!] \rho,\ [\![e_3]\!] \rho)) ([\![e_1]\!] \rho)$$

$$[\![rec\ x.e]\!] \rho = <\Upsilon(\lambda\phi. (\lambda <v,c>.(\lambda <v',c'>.<v', c'+c>)\circ v)([\![e]\!] \rho[\phi/x])), 1>$$

The model is intended to reflect the bahavior of programs that are compiled for a conventional von Neumann uniprocessor using static binding of free identifiers, call-by-value parameter discipline, a heap of activation records, and closures for function representation. To show how this intent is carried out, we explain several points about the semantic clauses.

First, the cost of invoking any primitive value or operation is 1. Intuitvely, this represents the cost of loading the value or fetching the instruction. Applying a primitive operation incurs an additional expense of 1 unit. Throughout, 1 is used abstractly for the concept "small constant no greater than some fixed amount". Thus, differences between memory access and instruction times are ignored in the model.

The cost of fetching the value of an identifier from the environment is also given as 1. In practice, the access time is proportional to the relative static level of the applied occurrence of the identifier if a static chain is traced to find the environment of definition. If a display is used, the access time is a small constant (the cost of an indexed indirect memory reference), but in that case the cost of maintaining the display must be accounted for. We have finessed the issue here by giving the access time as 1, noting that the model should be revised to give more accurate estimates for specific implementations of the environment.

The clause for function application indicates that the operator as well as all of the operands are evaluated before the operator is applied, corresponding to the call-by-value parameter discipline. Note that the cost of an application includes the cost of evaluating the operator, the cost of evaluating all of the operands, and the cost of computing the result of the actual application.

The cost of an abstraction is given as n, since this is a rough measure of the size of the activation record that must be set up. This may not be a good measure in practice. The author would be grateful for any suggestions on how to improve this aspect of the model. From the point of view of developing the basic theoretical machinery for reasoning about the complexity of functional programs, however, the extent to which such minor details of the model conform to any actual implementation is irrelevant.

The final clause, defining *rec*, seems more complicated than it really is. Intuitively, e is evaluated in $\rho[\phi/x]$, resulting in a value v and a cost c. For example, c is the cost of the activation record for e if e is an abstraction. When v (the "body" of the recursion) is applied, it produces a result v' and the cost of producing that result, c'. The result of a recursive call is therefore v', and the cost of the recursive call is $c'+c$.

The programming language was implemented by simply transliterating the semantics given here into ml [2]. The running model was invaluable as an aid in checking the axiomatic theory for conformance to the semantics. It was also helpful in formulating and refining the theory, helped us to correct errors in several of the example derivations, and in one case even exposed a serious omission in the semantics itself! Once the theory had been developed and "debugged" to our satisfaction, the task of checking it formally for consistency with the model was straightforward. We highly recommed having such a running model to anyone engaged in an experiment of formalization.

3. Axiomatization of the Model

As a first attempt to axiomatize this model, one might think of defining functions V and C giving the value and cost, respectively, of expressions. Doing this, one quickly discovers that this is too simple, because it fails to involve the environment.

The simplicity of the idea is too appealing to abandon it hastily, however. After all, in ordinary mathematics one often speaks of such nebulous quantities as "f(x)", leaving implicit the universal quantification over environments. There is, I am sure, a clean logical analysis of such locutions, but let's not worry about it just now. From a pragmatic point of view, we simply do not define any simplification of the term "x" to either a cost or a value. Proceeding in this manner gives us the following definitions.

$V(n) = n$

$C(n) = 1$

$V(T) = 1$

$C(T) = 1$

$V(F) = 0$

$C(F) = 1$

$V(+) = \lambda < v_1, v_2 > . < v_1 + v_2, 1 >$

$C(+) = 1$

similarly for other primitives

$V(x)$, $C(x)$ are irreducible.

$$V(e_0(e_1, \cdots, e_n)) = \mathit{valof}(V(e_0) {<} V(e_1), \cdots, V(e_n){>})$$

$$C(e_0(e_1, \cdots, e_n)) = C(e_0) + \sum_{i=1}^{n} C(e_i) + \mathit{costof}(V(e_0) {<} V(e_1), \cdots, V(e_n){>})$$

$$V(\lambda(x_1, \cdots, x_n).e) = \lambda {<} v_1, \cdots, v_n {>}. {<} V(e[v_1/x_1, \cdots, v_n/x_n]),$$
$$C(e[v_1/x_1, \cdots, v_n/x_n]){>}$$

$$C(\lambda(x_1, \cdots, x_n).e) = n$$

$$V(e_1 \rightarrow e_2,\ e_3) = V(e_1){=}1 \rightarrow V(e_2),\ V(e_3)$$

$$C(e_1 \rightarrow e_2,\ e_3) = V(e_1){=}1 \rightarrow C(e_1){+}\ C(e_2){+}1,\ C(e_1)\ C(e_3){+}1$$

$$V(\mathit{rec}\ x.e) = \mathbf{Y}(\lambda\phi.V(e[\phi/x]))$$

$$C(\mathit{rec}\ x.e) = 1$$

Unfortunately, this set of rules breaks down as soon as we try to apply a computed function, as in example **4.4** below. The reader should attempt to carry through that example using V and C in order to understand the exact nature of the difficulty. Other things that we tried in the process of inventing a formalism included the use of Hoare triples, direct manipulation of the semantic formulae, and a logic that had two separate sets of connectives for reasoning about costs and values.

All of these systems failed because they tried to cast everything solely in terms of costs and values. The key to a successful system was the realization that there are two kinds of costs involved: the cost of producing an entity, and the cost of using it. In an earlier version of the theory, we referred to the first of these simply as the entity's *cost*; the second was its *toll*. This two-level system works fine for describing simple functions, but breaks down as soon as we try to deal with higher-order functions, because we may need to know not only the cost of applying a function, but the cost of applying its result, the cost of applying the result of its application, and so forth ad infinitum.

The system we ultimately adopted results from the decomposition of the semantics into two domains of *Values* and *Tolls*. As in the denotational semantics, *Costs* are natural numbers. *Value* corresponds to *Val* stripped of its cost component, so that in the logic $Value = \mathbf{Z} + [Value^* \rightarrow Value]$. The cost computation role is delegated to tolls, which therefore belong to the domain $Toll = Cost + [Value^* \rightarrow Toll]$.

In the sequel, v_e denotes the value of the expression e, and t_e^j denotes a toll that is j levels of functional abstraction away from a simple cost. We sometimes refer to this as the j-toll of e. Notice that a given expression may be associated with tolls at many levels. For example, $v_+ = \lambda <n,m>.n+m$, $t_+^0 = 1$, and $t_+^1 = \lambda <n,m>.1$. Without more ado, we present the theory.

$\vdash v_n = n \wedge t_n^0 = 1$

$\vdash v_T \wedge t_T^0 = 1$

$\vdash \neg v_F \wedge t_F^0 = 1$

$\vdash v_+ = \lambda <x,y>.x+y \wedge t_+^0 = 1 \wedge t_+^1 = \lambda <x,y>.1$

... similarly for other primitives

$\vdash v_{e_1} \Rightarrow v_{e_1 \rightarrow e_2,\, e_3} = v_{e_2}$

$\vdash \neg v_{e_1} \Rightarrow v_{e_1 \rightarrow e_2,\, e_3} = v_{e_3}$

$\vdash v_{e_1} \Rightarrow t^0_{e_1 \rightarrow e_2,\, e_3} = t^0_{e_1} + t^0_{e_2} + 1$

$\vdash \neg v_{e_1} \Rightarrow t^0_{e_1 \to e_2, e_3} = t^0_{e_1} + t^0_{e_3} + 1$

$\vdash \forall\, j > 0.\ v_{e_1} \Rightarrow t^j_{e_1 \to e_2, e_3} = t^j_{e_2}$

$\vdash \forall\, j > 0.\ \neg v_{e_1} \Rightarrow t^j_{e_1 \to e_2, e_3} = t^j_{e_3}$

$\vdash v_{e_0(e_1, \cdots, e_n)} = v_{e_0} < v_{e_1}, \cdots, v_{e_n} >$

$\vdash t^0_{e_0(e_1, \cdots, e_n)} = t^0_{e_0} + \sum_{i=1}^{n} t^0_{e_i} + t^1_{e_0} < v_{e_1}, \cdots, v_{e_n} >$

$\vdash \forall\, j > 0.\ t^j_{e_0(e_1, \cdots, e_n)} = t^{j+1}_{e_0} < v_{e_1}, \cdots, v_{e_n} >$

$\vdash v_{\lambda(x_1, \cdots, x_n).e} = \lambda < v_1, \cdots, v_n > . v_{e[v_1/x_1, \cdots, v_n/x_n]}$

$\vdash t^0_{\lambda(x_1, \cdots, x_n).e} = n$

$\vdash \forall\, j > 0.\ t^j_{\lambda(x_1, \cdots, x_n).e} = \lambda < v_1, \cdots, v_n > . t^{j-1}_{e[v_1/x_1, \cdots, v_n/x_n]}$

$\vdash t^0_{v_e} = 1$

$\vdash \forall\, j > 0.\ t^j_{v_e} = t^j_e$

$\vdash v_{v_e} = v_e$

$\vdash v_{rec\ x.e} = \mathbf{Y}(\lambda < \phi > . e[\phi/x])$

The usual methods of induction can be used to reason about the value of recursive functions. Finally, we have an axiom and a pair of rules for making inductive inferences about tolls. The axiom concerns the 0-toll of a recursive expression.

$$\vdash t^0_{rec\ x.e} = 1$$

The first inference rule concerns the 1-toll of a recursive expression.

$$\frac{t^0_\phi = t^0_{rec\ x.e} \vdash P^f_{t^1_\phi} \Rightarrow P^f_{(\lambda < k > . k + t^0_e) \circ t^1_{e[\phi/x]}}}{\vdash P^f_{t^1_{rec\ x.e}}}$$

The hypothesis of the premiss could be more simply stated as $t^0_\phi = 1$, but this would mask its conformity with the next rule, which concerns the n-toll of a recursive expression, for $n > 1$.

$$\frac{\bigwedge_{i=0}^{n-1}(t^i_\Phi = t^i_{rec\ x.e}) \vdash P^f_{t^n_\Phi} \Rightarrow P^f_{t^n_{e[\Phi/x]}}}{\vdash P^f_{t^n_{rec\ x\ e}}}$$

The toll induction rules do not have an explicit base premiss because the base for the recursion (if any) is embedded in the body of the expression. Also note that the rules are partial in the sense that the conclusion holds only when the recursion terminates.

4. Some Examples

In this section we present a graded sequence of examples illustrating various apspects of the logic. None of the results is particularly surprising; the important point is that results such as corollary **4.4.1** cannot even be expressed in conventional terms. The early examples are carried out in great detail; the reader is asked to fill in more for herself in the later ones.

4.1. Elementary Application

$$\begin{aligned}\vdash t^0_{3+5} &= t^0_+ + t^0_3 + t^0_5 + t^1_+ <v_3, v_5> \\ &= 1 + 1 + 1 + (\lambda <x,y>.1)<3,5> \\ &= 3 + 1 = 4\end{aligned}$$

4.2. Variables

$$\begin{aligned}\vdash t^0_{x+5} &= t^0_+ + t^0_x + t^0_5 + t^1_+ <v_x, v_5> \\ &= 2 + t^0_x + (\lambda <x,y>.1)<v_x, 5> \\ &= 3 + t^0_x\end{aligned}$$

4.3. Abstraction

$$\vdash t^0_{(\lambda(x,y).\ x+y)(3,5)} = t^0_{\lambda(x,y).\ x+y} + t^0_3 + t^0_5 + t^1_{\lambda(x,y).\ x+y}<v_3,\ v_5>$$

$$= 2 + 1 + 1 + (\lambda<v_1,v_2>.t^0_{v_1+v_2})<3,5>$$

$$= 4 + t^0_{3+5} = 4 + 4 = 8$$

4.4. Higher-Order Functions

This example is in a sense both the motivation and the justification for our theory.

Let *twice* $\equiv \lambda(f).\lambda(x).f(f(x))$.

$$\vdash t^0_{(twice(a))(b)} = t^0_{twice(a)} + t^0_b + t^1_{twice(a)}<v_b>$$

$$= t^0_{twice} + t^0_a + t^1_{twice}<v_a> + t^0_b + (t^2_{twice}<v_a>)<v_b>$$

$$= 1 + t^0_a + (\lambda<v_1>.\ t^0_{\lambda(x).v_1(v_1(x))})<v_a> + t^0_b + (\lambda<v_1>.\ t^1_{\lambda(x).v_1(v_1(x))})<v_a><v_b>$$

$$= 1 + t^0_a + (\lambda<v_1>.1)<v_a> + t^0_b + (\lambda<v_1>.\lambda<v_2>.t^0_{v_1(v_1(v_2))})<v_a><v_b>$$

$$= 1 + t^0_a + 1 + t^0_b + t^0_{v_a(v_a(v_b))}$$

$$= 2 + t^0_a + t^0_b + t^0_{v_a} + t^0_{v_a(v_b)} + t^1_{v_a}<v_{v_a(v_b)}>$$

$$= 2 + t^0_a + t^0_b + 1 + t^0_{v_a} + t^0_{v_b} + t^1_{v_a}<v_b> + t^1_a<v_{v_a}<v_{v_b}>>$$

$$= 3 + t^0_a + t^0_b + 1 + 1 + t^1_a<v_b> + t^1_a<v_a<v_b>>$$

$$= 5 + t^0_a + t^0_b + t^1_a<v_b> + t^1_a<v_a<v_b>>$$

4.4.1. Corollary:

$$\vdash (\forall x,y.t^1_a<x> = t^1_a<y>) \Rightarrow t^0_{(twice(a))(b)} = 5 + t^0_a + t^0_b + 2t^1_a<v_b>$$

4.5. Theorem Composition

It is easy to show that

$$\vdash t^1_{\lambda(x).x} = \lambda<v_1>.1$$

From this we see that

$\vdash \forall\alpha,\beta.\ t^1_{\lambda(x).x}<\alpha> \ = \ t^1_{\lambda(x).x}<\beta>$

Hence, by the corollary to the previous exercise,

$$\begin{aligned}\vdash t^0_{(twice(\lambda(x).x))(b)} &= 5 + t^0_{\lambda(x).x} + t^0_b + 2t^1_{\lambda(x).x}<v_b> \\ &= 5 + 1 + t^0_b + 2(\lambda <v_1>.1)<v_b> \\ &= 8 + t^0_b\end{aligned}$$

4.6. Conditionals

$$\vdash v_{x<y} \ \Rightarrow \ t^0_{x<y \to x,\, y} = t^0_{x<y} + t^0_x + 1 \qquad (^*)$$

$$\vdash \neg v_{x<y} \ \Rightarrow \ t^0_{x<y \to x,\, y} = t^0_{x<y} + t^0_y + 1 \qquad (^{**})$$

Since

$$t^0_x \leq N \wedge t^0_y \leq N \vdash v_{x<y} \vee \neg v_{x<y},$$

we can infer

$$t^0_x \leq N \wedge t^0_y \leq N \vdash t^0_{x<y \to x,\, y} \leq t^0_{x<y} + N + 1 \qquad (^{***})$$

from (*) and (**) using the case analysis rule, arithmetic, and transitivity of $\Rightarrow$.

A straightforward derivation establishes

$$t^0_x \leq N \wedge t^0_y \leq N \vdash t^0_{x<y} \leq 2 + 2N$$

Combining this result with (***) gives

$$t^0_x \leq N \wedge t^0_y \leq N \vdash t^0_{x<y \to x,\, y} \leq 3N + 3$$

4.7. Criteria for Algorithm Selection

In this example, we demonstrate the use of our complexity logic to choose between two programs on the basis of their relative efficiency. Using an argument similar to that of the previous example, we first establish that

$$v_{a<b} \vee \neg v_{a<b} \vdash t^0_{a<b \to a,\, b} \geq 3 + t^0_a + t^0_b + min(t^0_a,\, t^0_b)$$

Similarly, it is easy to show that

$$v_{a<b} \vee \neg v_{a<b} \vdash t^0_{(\lambda(x,y).x<y \to x,\, y)(a,b)} = 8 + t^0_a + t^0_b$$

Hence, we conclude that the more complex form, which avoids evaluating a and b more than once, has an additional overhead of 5. Whenever $min(t_a^0, t_b^0)$ exceeds this amount, the second program is preferable from an efficiency standpoint.

4.8. Exact Analysis of Recursive Functions

Let

$! \equiv rec\ f.\ \lambda(n).n \leq 0 \rightarrow 1,\ n \times f(n-1),$
and assume that $t_\phi^1 = \lambda <v_1>.14 \times max(v_1,0) + 7$. Now,

$$\vdash (\lambda <k>.k + t^0_{\lambda(n).n \leq 0 \rightarrow 1,\ n\times f(n-1)}) \circ t^1_{\lambda(n).n \leq 0 \rightarrow 1,\ n\times \phi(n-1)}$$
$$= (\lambda <k>.k + 1) \circ \lambda <v_1>.t^0_{v_1 \leq 0 \rightarrow 1,\ v_1 \times \phi(v_1 - 1)}$$
$$= \lambda <v_1>.t^0_{v_1 \leq 0 \rightarrow 1,\ v_1 \times \phi(v_1 - 1)} + 1$$

From here we proceed by case analysis on the conditional.

4.8.1. Lemma:

$$\vdash v_1 \leq 0 \Rightarrow t^0_{v_1 \leq 0 \rightarrow 1,\ v_1 \times \phi(v_1 - 1)} + 1 = t^0_{v_1 \leq 0} + t^0_1 + 1 + 1$$
$$= 4 + 1 + 2 = 7$$
$$= 14 \times max(v_1, 0) + 7$$

4.8.2. Lemma:

$$\vdash \neg v_1 \leq 0 \Rightarrow t^0_{v_1 \leq 0 \rightarrow 1,\ v_1 \times \phi(v_1 - 1)} + 1$$
$$= t^0_{v_1 \leq 0} + t^0_{v_1 \times \phi(v_1 - 1)} + 1 + 1$$
$$= 4 + t^0_{v_1} + t^0_\times + t^0_{\phi(v_1 - 1)} + t^1_\times <v_{v_1}, v_{\phi(v_1-1)}> + 2$$
$$= 6 + 1 + 1 + t^0_\phi + t^0_{v_1 - 1} + t^1_\phi <v_{v_1 - 1}> + 1$$

At this point we invoke the assumption that $t^0_\phi = t^0_! = 1$. From this and the induction hypothesis, the preceeding formula becomes

$$= 9 + 1 + 4 + (\lambda<v_1>.14\times max(v_1,0) + 7)<v_1-1>$$

$$= 14 + 14\times max(v_1-1,0) + 7$$

$$= 14\times max(v_1,0) + 7$$

where the last line depends on the hypothesis that $\neg v_1 \leq 0$. The pair of lemmas allows us to conclude that

$$\vdash \lambda<v_1>.t^0_{v_1 \leq 0 \rightarrow 1,\ v_1\times\phi(v_1-1)} + 1 = \lambda<v_1>.14\times max(v_1,0) + 7$$

By the first toll induction rule, therefore,

$$t^1_! = \lambda<v_1>.14\times max(v_1,0) + 7$$

4.8.3. Corollary:

$$\vdash t^0_{!(a)} = t^0_! + t^0_a + t^1_!<v_a>$$

$$= 1 + t^0_a + 14\times max(v_a,0) + 7$$

$$= 14\times max(v_a,0) + t^0_a + 8$$

This prediction has been checked against the running model and found to be correct.

4.9. Asymptotic Analysis of Recursive Functions

Let $fib \equiv rec\ f.\ \lambda(n).n \leq 1 \rightarrow 1, f(n-1) + f(n-2)$. A casual inspection of this algorithm suggests that it should be exponentially bounded. Stated precisely, we expect to be able to prove that

$$\exists\ \alpha \geq 0.\ t^1_{fib} \leq \lambda<v_1>.\alpha\times 2^{max(v_1-1,0)},$$

where $\leq$ is extended to functions pointwise. This formula is proved here. In the following problem we show how intuition about the complexity of a program can be strengthened by inspecting the proof.

Given

$\vdash \exists\ \alpha \geq 0.\ t^1_\phi \leq \lambda <v_1>.\alpha \times 2^{max(v_1-1,0)}$

as our inductive hypothesis, we make the inductive step

$$\vdash (\lambda <k>.\ k\ +\ t^0_{\lambda(n).n \leq 1 \to 1,\ f(n-1)+f(n-2)}) \circ t^1_{\lambda(n).n \leq 1 \to 1,\ \phi(n-1)+\phi(n-2)}$$

$$= (\lambda <k>.K\ +\ 1) \circ \lambda <v_1>.t^0_{v_1 \leq 1 \to 1,\ \phi(v_1-1)+\phi(v_1-2)}$$

$$= \lambda <v_1>.t^0_{v_1 \leq 1 \to 1,\ \phi(v_1-1)+\phi(v_1-2)}\ +\ 1$$

As before, we proceed by case analysis on the conditional.

4.9.1. Lemma:

$$\vdash v_1 \leq 1 \implies t^0_{v_1 \leq 1 \to 1,\ \phi(v_1-1)+\phi(v_1-2)} + 1 = t^0_{v_1 \leq 1} + t^0_1 + 1 + 1$$

$$= 7 \leq 7 \times 2^{max(v_1-1,0)}$$

4.9.2. Lemma:

$$\vdash \neg v_1 \leq 1 \implies t^0_{v_1 \leq 1 \to 1,\ \phi(v_1-1)+\phi(v_1-2)} + 1$$

$$= t^0_{v_1 \leq 1} + t^0_{\phi(v_1-1)+\phi(v_1-2)} + 1 + 1$$

$$= 6 + t^0_+ + t^0_{\phi(v_1-1)} + t^0_{\phi(v_1-2)} + t^1_+ <v_{\phi(v_1-1)}, v_{\phi(v_1-2)}>$$

$$= 8 + t^0_\phi + t^0_{v_1-1} + t^1_\phi <v_{v_1-1}> + t^0_\phi + t^0_{v_1-2} + t^1_\phi <v_{v_1-2}>$$

$$= 18 + t^1_\phi <v_1-1> + t^1_\phi <v_1-2>$$

$$\leq 18 + \alpha \times 2^{max(v_1-2,0)} + \alpha \times 2^{max(v_1-3,0)}$$

by the induction hypothesis.

$$= 18 + 3\alpha \times 2^{max(v_1-3,0)}$$

choosing $\alpha = 18$, this is then

$$= 4 \times 18 \times 2^{max(v_1-3,0)}$$

$$= 18 \times 2^{max(v_1-1,0)}$$

From the two lemmas, we conclude

$$\vdash \lambda < v_1 > . t^0_{v_1 \leq 1 \to 1,\ \phi(v_1 - 1) + \phi(v_1 - 2)} + 1 \leq \lambda < v_1 > . 18 \times 2^{max(v_1 - 1, 0)}$$

By toll induction,

$$\vdash t^1_{fib} \leq \lambda < v_1 > . 18 \times 2^{max(v_1 - 1, 0)}$$

which is actually stronger than the theorem we set out to prove; we have a conservative estimate for α, namely 18.

4.10. Extracting Complexity Estimates from Proofs

When this relationship is used to predict the performance of *fib,* we find that, although the relationship holds, the bound is far from tight, as the following table shows:

	fib(a)	
a	predicted	measured
0	20	9
1	20	9
2	38	34
3	74	59
4	146	109
5	290	184
6	578	309

An inspection of the proof suggests a refined complexity estimate as follows. The value of $fib(v_1)$ is built up in this program by adding 1s from cases where $v_1 \leq 1$, and adding partial sums from recursive calls when $\neg v_1 \leq 1$. Each base case incurs an expense of 7 units; each non-base case costs 18 units. It is easy to see that there are $v_{fib(v_1)}$ base cases, and $v_{fib(v_1)} - 1$ non-base cases. This reasoning leads to a refined (in fact, exact) estimate:

$$t^1_{fib} = \lambda < v_1 > . 7 \times v_{fib(max(v_1, 0))} + 18 \times (v_{fib(max(v_1, 0))} - 1)$$

The necessary modifications to the proof are straightforward.

This example suggests a method for deriving the complexity of a recursive function, which we have found to be extremely useful. Essentially, we massage the consequent of the premiss of the toll induction rule until the antecedent (the "invariant") simply falls out.

5. Data Structures

Having illustrated all of the basic techniques for analyzing the complexity of higher-order programs, we now show how these methods are used to choose between two possible representations for sequences. Throughout this section, we shall simply state the results, leaving the detailed proofs to the interested reader. In no case is any proof more complicated than those we have already shown.

Our simple functional language does not have any data structuring facilities, so we must compose our own. Of course, a given data abstraction can be represented in many ways, and it is incumbent upon the programmer to choose a representation that will perform well (at least in the context of the intended application).

Consider the problem of choosing a representation for sequences in our simple language. One possibility is to represent the sequence by a pair consisting of a natural number and a mapping from natural numbers to elements of the sequence. A second possibility is to represent the sequence recursively as being either empty or a pair consisting of an element and a sequence.

5.1. Pairs

In either case, we shall need pairing and projections. We define these as follows.

$pair \equiv \lambda(x,y).\ \lambda(i).\ i \le 0 \rightarrow x,\ y$

$p1 \equiv \lambda(p).\ p(0)$

$p2 \equiv \lambda(p).\ p(1)$

For future reference, we record here the relevant facts about these operations.

$t^0_{pair} = 2 \wedge t^1_{pair} = \lambda <v_1,v_2>.\ 1$

$t^0_{pair(a,b)} = t^0_a + t^0_b + 3 \wedge t^1_{pair(a,b)} = \lambda <v_1>.\ 6$

$t^0_{pj(x)} = 3 + t^0_x + t^1_x <v_j> \quad (j=1,2)$

This last quantity is $= 9 + t^0_x$ provided that x is constructed using the *pair* operation.

5.2. Mapping Representation of Sequences

We shall consider first the representation of sequences by a (length,mapping) pair. The necessary operations are defined as follows.

$null \equiv \lambda(s).\ p1(s)=0$

$hd \equiv \lambda(s).\ (p2(s))(p1(s))$

$tl \equiv \lambda(s).\ pair(p1(s)-1,\ p2(s))$

$cons \equiv \lambda(x,s).\ (\lambda(n).\ pair(n,\ \lambda(m).\ m=n \rightarrow x,\ (p2(s))(m)))(p1(s) + 1)$

$nil \equiv pair(0,\ \lambda(n).0)$

$length \equiv \lambda(s).\ p1(s)$

Before we work out the relevant facts about the performance of these operations, we shall present the recursive, or "tuple" representation. By delaying the analysis, we hope to convince the reader of the difficulty of deciding which representation is best without doing the analysis. The reader whose intuition about higher-order programs is strong enough to see the outcome in advance is doubtless able to construct a more complicated example where her intuition will fail (regardless of her sex).

5.3. Tagged Unions

Before we can proceed with the tuple representation, we introduce the operations for tagged unions as follows.

$in1 \equiv \lambda(x).\ pair(1,x)$

$in2 \equiv \lambda(x).\ pair(2,x)$

$is1 \equiv \lambda(u).\ p1(u)=1$

$is2 \equiv \lambda(u).\ p1(u)=2$

$out \equiv \lambda(u).\ p2(u)$

The relevant facts about these operations are the following.

$\vdash t^0_{inj(x)} = t^0_x + 6$

$\vdash t^0_{isj(u)} = t^0_u + 14$

$\vdash t^0_{out(u)} = t^0_u + 11$

5.4. Tuple Representation of Sequences

The operations for the tuple representation are defined as follows.

$null \equiv is1$

$hd \equiv \lambda(s).\ p1(out(s))$

$tl \equiv \lambda(s).\ p2(out(s))$

$cons \equiv \lambda(x,s).\ in2(pair(x,s))$

$nil \equiv in1(0)$

$length \equiv rec\ f.\ \lambda(s).\ null(s) \rightarrow 0,\ 1 + f(tl(s))$

5.5. Comparing the Representations

The relative efficiency of these two representations is not immediately apparent. As might be expected, each has its advantages. The results of the analysis are outlined in the following table.

	cost	
operation	mapping	tuple
null(*s*)	$t_s^0 + 14$	$t_s^0 + 14$
hd(*s*)	$21 + t_s^0 + t_s^2\langle 1\rangle\langle v_{p1(v_s)}\rangle$	$t_s^0 + 22$
tl(*s*)	$t_s^0 + 27$	$t_s^0 + 22$
cons(*x*,*s*)	$t_x^0 + t_s^0 + 21$	$t_x^0 + t_s^0 + 13$
nil	5	7
length(*s*)	$t_s^0 + 11$	$t_s^0 + 44 \times v_{length(s)} + 19$

From the table we see that the tuple operations tend to be at least as efficient as the mapping operations, the exceptions being *nil* and *length*. Only the latter is significant enough to make the mapping representation worth considering for some applications.

Three points about this exercise are salient. First, it took the author considerably less than an hour to derive all of the tabulated results. Experimental measurements leading to a similar table would have taken considerably more time and effort, to say nothing of the machine resources consumed. Second, the constant differences are sufficiently small compared to the resolution of timing information available on most systems that conclusive measures would likely require running programs involving many thousands of operations. To write a test program involving so many operations without using recursion is impractical. But if recursion is used, then its contribution to the total running time must be factored out, which requires some kind of analysis anyway. Finally, we seriously doubt our ability to make accurate estimates of the complexity by simply eyeballing the code. A thorough analysis or set of measurements is imperative if we are to have any confidence in our knowledge of the efficiency of our programs.

These considerations encourage us to believe that our analytic theory offers significant practical advantages over experimental methods. Our belief must be tempered, however, by the usual caution against relying on any single method, especially when the method involves a human agent. In short, the author made a clerical error in the analysis of *length* in the tuple representation, resulting in a slightly low estimate of $t_s^0 + 43\times v_{length(s)} + 18$. The error was revealed by comparing this prediction against the model.

5.6. Another Look at *hd*

The entry for *hd* in the mapping representation demands explanation. It is the only example we have seen so far of a complexity that is not a simple expression consisting of constants and elementary properties of the arguments. Put simply, the problem is that *tl* does not make the mapping any simpler, it just decrements the length. Consequently, the mapping portion of *cons*(*hd*(*s*),*tl*(*s*)) is significantly more complex than that of the original *s*, even though the two sequences behave alike as values. The upshot is that the complexity of *hd* in the mapping representation depends on history - the sequence of *cons* and *tl* operations used to construct its argument.

Let us carry through some calculations concerning *hd* to see how this situation arises. We begin by deriving the tabulated formula.

$$\begin{aligned}\vdash t_{hd(s)}^0 &= t_{hd}^0 + t_s^0 + t_{hd}^1\langle v_s\rangle \\ &= 1 + t_s^0 + t_{(p2(v_s))(p1(v_s))}^0 \\ &= 1 + t_s^0 + t_{p2(v_s)}^0 + t_{p1(v_s)}^0 + t_{p2(v_s)}^1\langle v_{p1(v_s)}\rangle\end{aligned}$$

$$= 1 + t_s^0 + 10 + 10 + t_{p2}^2 <v_s><v_{p1(v_s)}>$$

$$= 21 + t_s^0 + (\lambda <v_1>.\ t^1_{v_1(1)})<v_s><v_{p1(v_s)}>$$

$$= 21 + t_s^0 + t^1_{v_s(1)}<v_{p1(v_s)}>$$

$$= 21 + t_s^0 + t_s^2<1><v_{p1(v_s)}>$$

This formula could be further simplified if we had a simple characterization of t_s^2 for an arbitrary sequence s. In order to develop such a characterization, we must consider all of the ways that we can construct a sequence, i.e. all of the operations that produce a sequence as their result. These are *cons*, *tl*, and *nil*. We shall begin our analysis with *cons*.

$$\vdash t^2_{cons(x,s)} = t^3_{cons}<v_x, v_s>$$

$$= (\lambda <v_1,v_2>.\ t^2_{(\lambda(n).\ pair(n,\lambda(m).\ m=n \rightarrow v_1,\ (p2(v_2))(m)))(p1(v_2)+1)})<v_x, v_s>$$

$$= t^2_{(\lambda(n).\ pair(n,\lambda(m).\ m=n \rightarrow v_x,\ (p2(v_s))(m)))(p1(v_s)+1)}$$

$$= t^3_{\lambda(n).\ pair(n,\lambda(m).\ m=n \rightarrow v_x,\ (p2(v_s))(m))}<v_{p1(v_s)+1}>$$

$$= t^2_{pair(v_{p1(v_s)+1},\ \lambda(m).\ m=v_{p1(v_s)+1} \rightarrow v_x,\ (p2(v_s))(m))}$$

$$= t^3_{pair}<v_{p1(v_s)+1},\ v_{\lambda(m).\ m=v_{p1(v_s)+1} \rightarrow v_x,\ (p2(v_s))(m)}>$$

$$= t^2_{\lambda(i).\ i \le 0 \rightarrow v_{p1(v_s)+1},\ v_{\lambda(m).\ m=v_{p1(v_s)+1} \rightarrow v_x,\ (p2(v_s))(m)}}$$

$$= \lambda <v_1>.\ t^1_{v_1 \le 0 \rightarrow v_{p1(v_s)+1},\ v_{\lambda(m).\ m=v_{p1(v_s)+1} \rightarrow v_x,\ (p2(v_s))(m)}}$$

From here we proceed by case analysis on the conditional, as usual.

5.6.1. Lemma:

$$\vdash v_1 \le 0 \implies t^1_{v_1 \le 0 \rightarrow v_{p1(v_s)+1},\ v_{\lambda(m).\ m=v_{p1(v_s)+1} \rightarrow v_x,\ (p2(v_s))(m)}} = t^1_{p1(v_s)+1} = \bot$$

5.6.2. Lemma:

$$\vdash \neg v_1 \leq 0 \Rightarrow t^1_{v_1 \leq 0 \rightarrow v_{p1(v_s)+1},\ v_{\lambda(m).\ m = v_{p1(v_s)+1} \rightarrow v_x,\ (p2(v_s))(m)}}$$

$$= t^1_{\lambda(m).\ m = v_{p1(v_s)+1} \rightarrow v_x,\ (p2(v_s))(m)}$$

$$= \lambda < v_2 >.\ t^0_{v_2 = v_{p1(v_s)+1} \rightarrow v_x,\ (p2(v_s))(v_2)}$$

Now, this tells us that we're in trouble if we ever ask for $t^2_{cons(x,s)} <0>$. Looking back at the complexity of $hd(s)$, however, we see that we are only really interested in $t^2_{cons(x,s)} <1>$. So,

$$\vdash t^2_{cons(x,s)} <1> = \lambda < v_2 >.\ t^0_{v_2 = v_{p1(v_s)+1} \rightarrow v_x,\ (p2(v_s))(v_2)}$$

Again, we proceed by case analysis.

5.6.3. Lemma:

$$\vdash v_2 = v_{p1(v_s)+1} \Rightarrow t^0_{v_2 = v_{p1(v_s)+1} \rightarrow v_x,\ (p2(v_s))(v_2)}$$

$$= t^0_{v_2 = v_{p1(v_s)+1}} + t^0_{v_x} + 1$$

$$= 6$$

5.6.4. Lemma:

$$\vdash \neg v_2 = v_{p1(v_s)+1} \Rightarrow t^0_{v_2 = v_{p1(v_s)+1} \rightarrow v_x,\ (p2(v_s))(v_2)}$$

$$= t^0_{v_2 = v_{p1(v_s)+1}} + t^0_{(p2(v_s))(v_2)} + 1$$

$$= 4 + t^1_{p2(v_s)} < v_2 > + 1$$

$$= 5 + t^2_{p2} < v_s > < v_2 >$$

At this point, we can see that there is little purpose in trying to carry the analysis further, because a comparison of the above formula to the fourth line in the derivation of $t^0_{hd(s)}$ reveals that we have arrived at a subproblem with the same structure as the original problem. This would not cause us any difficulty if we were able to induct, but we can't without making addtional assumptions about the pattern of s's construction.

Perhaps the lesson to be learned from all this is that the price one pays for first-class functions is first-class complexities. The latter are not nearly so attractive as the former.

6. Higher-Order Recursive Functions

Our final result concerns the analysis of higher-order recursive functions. For purposes of discussion, we shall look at the higher-order function *map* mentioned in the introduction.

$map \equiv rec\ m.\ \lambda(f).\ \lambda(s).\ null(s) \rightarrow s,\ cons(f(hd(s)),\ (m(f))(tl(s)))$
We are interested in the 0-, 1-, and 2-tolls of *map*. The 0-toll is just 1, by the axiom. A straightforward argument using the 1-toll induction rule shows that

$\vdash t^1_{map} = \lambda <v_1>.\ 2$
To find t^2_{map}, we massage the consequent of the premiss to the n-toll induction rule, hoping to discover the invariant.

$$\vdash t^2_{\lambda(f).\ \lambda(s).\ null(s) \rightarrow s,\ cons(f(hd(s)),(\phi(f))(tl(s)))}$$
$$= \lambda <v_1>.\ \lambda <v_2>.\ t^0_{null(v_2) \rightarrow v_2,\ cons(v_1(hd(v_2)),(\phi(v_1))(tl(v_2)))}$$
We proceed by case analysis on the conditional. We simply state the relevant lemmas here, leaving the details of the derivations to the interested reader.

6.1. Lemma:

$\vdash null(v_2) \Rightarrow t^0_{null(v_2) \rightarrow v_2,\ cons(v_1(hd(v_2)),(\phi(v_1))(tl(v_2)))} = 2 + t^0_{null(v_2)}$
Before stating the second lemma, we make some definitions.

$\alpha \equiv t^0_{null(v_2)}$

$\beta = t^0_{cons}$

$\gamma = t^0_{hd(v_2)}$

$\delta = t^1_{cons} < v_{v_1(hd(v_2))}, v_{(\phi(v_1))(tl(v_2))} >$

$\epsilon = t^0_{tl(v_2)}$

6.2. Lemma:

$$\vdash \neg null(v_2) \Rightarrow t^0_{null(v_2) \rightarrow v_2,\, cons(v_1(hd(v_2)),(\phi(v_1))(tl(v_2)))}$$

$$= 6 + \alpha + \beta + \gamma + \delta + \epsilon + t^1_{v_1} < v_{hd(v_2)} > + t^2_{\phi} < v_1 >< v_{tl(v_2)} >$$

Unless we make some assumptions about the sequence operations, it is difficult to formulate an induction hypothesis from these lemmas that would allow us to collapse the conditional and complete the induction. In the tuple representation of sequences, the quantities α-ϵ are all constants, so that the last formula simplifies to

$$74 + t^1_{v_1} < v_{hd(v_2)} > + t^2_{\phi} < v_1 >< v_{tl(v_2)} >$$

A successful induction hypothesis is then

$$t^2_{\phi} = \lambda < v_1 >. \lambda < v_2 >. 74 \times length(v_2) + 17 + \sum_{i=1}^{length(v_2)} t^1_{v_1} < i^{th}(v_2) >$$

where i^{th} has the obvious mathematical meaning, and the instances of *length* refer to the value of the *length* function. If $t^1_{v_1}$ is a constant function having value η, as in the case of *succ* ($\eta = 4$), then this can be further simplified to

$$t^2_{\phi} = \lambda < v_1 >. \lambda < v_2 >. (74 + \eta) \times length(v_2) + 17$$

The reader is invited to attempt to formulate analogous induction hypotheses for the mapping representation, where γ is not a constant.

7. Conclusions

Those interested in efficiency have long looked askance at functional programming, believing that functional programs are inherently less efficient than imperative ones. It may be more accurate, however, to say that it is harder to reason about the complexity of functional programs, and that writing efficient functional programs is thereby harder.

Advocates of functional programming, on the other hand, stress that the promise of transformational programming can only be realized in a language where reasoning about functionality is easy. In order to be effective, however, the effect of a transformation on the efficiency of the program must be determined, and this may be harder in a functional language.

Our experience suggests that the views of both camps trivialize a subtle relationship between complexity and functionality. Expressions in functional languages behave like functions, making it easy to reason about them using the mathematics of functions. By obscuring the relationship of programs to hardware, they make reasoning about performance harder. On the other hand, the constructs in imperative languages behave like hardware, making it easy to reason about them using the mathematics of machine performance. The price paid is that it's then harder to understand programs functionally.

One cannot help but wonder whether this relationship is an accident of circumstance, or if it reflects a universal law of computing. A good argument can be made for the latter view, as follows. In any notation, the number of expressions denoting a given partial recursive function is denumerably infinite. An algorithm for evaluating expressions cannot possibly lead to the same complexity for all of the expressions that denote a given function. Therefore, the logical rules governing functional equivalence cannot be complexity-preserving. A suitably formalized version of this is easily proved using basic methods of recursive function theory [4].

Philosophically, these considerations suggest that many, if not most, of our conceptualizations are in conflict with one another linguistically. This in turn suggests that we redirect our energies away from the attempt to design ideal notations for general-purpose programming, and toward the design of good notations for individual aspects of programming, along with methods for moving among them.

8. Extensions

Before we can claim to have a well-developed theory of the complexity of higher-order programs, our methods must be extended along several dimensions. The logical properties of other important parameter disciplines, such as call-by-name and call-by-need, must be determined. Once we understand a number of parameter disciplines in isolation, we should then merge them into a single theory that allows reasoning about programs in which the parameter discipline to be used in a given instance is specified by the programmer. This would allow us to develop guidelines on how to make effective use of annotations for controlling the evaluation of functional programs [1].

F. Warren Burton has begun a study of the space complexity of functional programs, and we believe that our techniques should be useful there, as well. An especially intriguing problem is to characterize the interaction between storage and time when garbage collection is considered.

Our long-term goal is to develop methods for analyzing the parallel complexity of functional programs. Other areas of interest include the complexity of non-deterministic and distributed higher-order programs, and the application of our methods to program transformation. References

1. F. W. Burton, Annotations to control parallelism and reduction order in the distributed evaluation of functional programs, *ACM Trans. Program. Lang.*

Syst. 6, 2 (April 1984), 159-174.

2. M. J. Gordon, A. J. Milner and C. P. Wadsworth, Edinburgh LCF, in *Lecture Notes in Computer Science, no. 78,* Springer-Verlag, Berlin, 1979.

3. R. E. Milne and C. Strachey, *A Theory of Programming Language Semantics,* John Wiley, 1976.

4. H. Rogers, *Theory of Recursive Functions and Effective Computability,* McGraw-Hill, 1967.

5. D. S. Scott and C. Strachey, Toward a Mathematical Semantics for Computer Languages, in *Proc. Symposium on Computers and Automata,* J. Fox (ed.), Polytechnic Institute of New York, New York, 1971, 19-46.