University of London
Imperial College of Science, Technology and Medicine
Department of Computing

Calculi for Time Analysis of Functional Programs

David Sands

A thesis submitted for the degree of
Doctor of Philosophy of the University of London
and for the Diploma of the Imperial College

September 1990

Abstract

Techniques for reasoning about extensional properties of functional programs are well-understood, but methods for analysing the underlying intensional, or operational properties have been much neglected. This thesis presents the development of several calculi for time analysis of functional programs.

We focus on two features, higher-order functions and lazy evaluation, which contribute much to the expressive power and semantic elegance of functional languages, but serve to make operational properties more opaque.

Analysing higher-order functions is problematic because complexity is dependent not only on the cost of computing, but also on the cost of *applying* function-valued expressions. Techniques for statically deriving programs which compute time-cost in the presence of arbitrary higher-order functions are developed. The key to this process is the introduction of syntactic structures called *cost-closures*, which enable intensional properties to be carried by functions. The approach is formalised by the construction of an appropriate cost-model, against which the correctness of the derivation is proved. A specific factorisation tactic for reasoning about higher-order functions out of context is illustrated.

Reasoning about lazy evaluation (*i.e.* call-by-name, or more usually, call-by-need) is problematic because the cost of evaluating an expression cannot be understood simply from the costs of its sub-expressions. A direct calculus for reasoning about a call-by-name language with lazy lists is derived from a simple operational model. In order to extend this calculus with a restricted form of equational reasoning, a non-standard notion of operational approximation called *cost-simulation* is developed, by analogy with *(bi)simulation* in CCS.

The problem with calculi of the above form, based directly on an operational model, is that they do not yield a *compositional* description of cost, and cannot model *lazy evaluation* (graph-reduction) easily. We show how a description of the *context* in which a function is evaluated can be used to parameterise two types of time-equation: *sufficient-time* equations and *necessary-time* equations, which together provide bounds on the exact time-cost of lazy evaluation. This approach is extended to higher-order functions using a modification of the cost-closure technique.

Acknowledgements

I am indebted to Chris Hankin, whose supervision and encouragement were invaluable to the evolution and completion of this thesis.

I have benefited greatly from numerous discussions and direct contributions from my colleagues and good friends Jesper Andersen and Sebastian Hunt. Many Thanks. I have also benefited from the research environment at Imperial, and in particular I would like to thank Geoff Burn, Tony Field, Pete Harrison, Thomas Jensen and Bent Thomsen as well as many other members of the Theory and Formal Methods and Functional Programming sections, for their various inputs. Thanks to Paul Taylor for TEXnical assistance, and to Anita Sasson for her meticulous proof-reading. Outside IC, I would like to thank Daniel Le Métayer for his constant interest and encouragement, Richard Bird for some useful ideas and advice, and Torben Mogensen for various helpful suggestions.

This work was funded in part by a studentship from the Science and Engineering Research Council of Great Britain, and ESPRIT project BRA 3124, Semantique. Thanks to Big Ron, Roony the House, Kosh the Dosh and The DB for miscellaneous(!) financial support.

Thanks to my mum and dad for providing a frequent (and often windy) haven from the grime of London life. Most of all thanks to Anita, not just for putting up with me, but for giving me the encouragement that I needed.

Dave Sands
September 1990

Contents

List of Figures

Chapter 1

Introduction

Proponents of functional programming provide some compelling arguments as to the expressive power of modern functional languages [Tur81, Hug89, Hud89]. Much of this expressive power is due to the use of data-abstraction facilities, higher-order functions, and lazy evaluation. Data abstraction provides modularity and clarity, and, via static typing, a degree of security. Higher-order functions (functions treated as first-class objects) provide an elegant structuring mechanism for building programs, and lazy evaluation (call-by-need together with delayed evaluation of data-structures) supports programming styles in which "infinite" data-structures are manipulated, and, to some degree, frees the programmer from concern with underlying operational issues.

Although many programming languages contain (some of) the above features, it is the underlying mathematical tractability of functional languages which is responsible for much of the interest in functional programming. The phrase *referential transparency* is used to describe a key mathematical property of functional programs: the ability to universally substitute equals for equals. The importance of referential transparency is that simple, but powerful equational reasoning (akin to "everyday mathematics") may be used to reason formally about programs.

These properties make functional languages particularly suited to activities such as *program transformation*, which seeks to harmonise the conflicting requirements of program clarity and efficiency by the synthesis of correct and efficient programs from clear initial solutions. There are various manifestations of this paradigm, from viewing transformation as a mathematical programming activity [Mee86, Bir86], to the construction of prototype systems to partially mechanise the formal development of correct software [DHK$^+$89, PS83]. In addition to the syntactic manipulation of programs, semantics-based techniques for determining sound optimisations of functional programs, such as strictness analysis, have received much attention [AH87].

Another significant area of interest is in the execution of functional programs on novel parallel architectures, by the exploitation of the implicit parallelism fostered by referential transparency.

The suitability of functional programs for these activities is largely due to the ease with which the *extensional* properties of a program are understood—above all the ability to show that operations on programs preserve meaning. Prominent in the study of algorithms in general, and central to formal activities such as program transformation and parallelization, are questions of efficiency, *i.e.* the running-time and space requirements of programs. These are *intensional* properties of a program—properties of *how* the program computes, rather that *what* it computes. However the study of intensional properties is not immediately amenable to the algebraic methods with which extensional properties are so readily explored, and very little attention has been given to the formal study of the computational cost of algorithms expressed as functional programs. Furthermore, the *declarative* emphasis of functional programs, together with some of the features that afford expressive power, namely higher-order functions and lazy evaluation, serve to make intensional properties *more opaque*.

In this thesis we intend to address some of the problematic issues that arise in the time-analysis of functional programs for which traditional methods are inadequate. We explore calculi for expressing the time-cost of programs with higher-order functions, and programs employing non-strict evaluation orders. In the remainder of this introduction we discuss the background, review related work, and outline the structure of the thesis.

1.1 Approaches to Time-Analysis

One way to measure the performance of a program is to benchmark it on some "suitably chosen" set of inputs, and observe the performance directly (the stop-watch approach) or via appropriate profiling tools. Our focus is not on this experimental approach, but on more *analytic* methods. To analyse time-cost analytically, one attempts to express properties of the running time (such as worst-case or expected time) by using mathematical techniques. Whilst to some extent the experimental and analytic techniques are complementary, the latter approach is attractive since it leads to a more informed judgement of an algorithms performance and has the possibility of being influential *during* the software-design process, for example by guiding program transformation steps, or by assisting in the choice of annotations for parallel evaluation.

The *time complexity* of a program is usually taken to mean a function mapping the "size" of the input to the worst-case time required. What is meant by "time" is not the number of seconds on a particular machine, but a more abstract discrete count of certain evaluation steps or operations.

Micro vs. Macro Descriptions

How we choose to represent time-steps roughly divides such analyses into two categories: *macro-analysis* and *micro-analysis*. Macro-analysis expresses time-complexity in terms of a dominant operation, or operations of an algorithm. For example, sorting algorithms are characterised by the number of comparisons used, array manipulation by the number of arithmetic operations—see [AHU74] for many classic examples. In a micro-analysis we seek to express complexity in terms of the number of each type of elementary operation performed, such as Knuth's use of a hypothetical machine MIX [Knu68] to express complexity in terms of individual machine instructions executed. More abstract notions of micro-analysis are possible: for one example see [Coh82].

1.2 The Analysis of Imperative Programs

Much of the work on the analysis of algorithms was initiated by Knuth. The three volumes of *The Art of Computer Programming* [Knu68, Knu69, Knu73] contain a great many program analyses, and illustrate many of the fundamental mathematical tools necessary. A good introduction to the analysis of time-complexity can be found in [AHU74, AHU82]. The focus of these works is the study of specific important algorithms, rather than general techniques. The time-analysis of algorithms expressed in a simple imperative language begins by intuitively clear but informal construction of time-equations, where loops lead informally to the summation-terms, and recursive procedures lead to the construction of recurrence equations. These informal derivations are easily justified because the algorithms are expressed in an imperative style which is "operationally declarative". This process can however be formalised using Floyd-Hoare logics, either by adding a time-count into the program (*e.g.* [Weg76b]) and using the standard proof techniques directly, or by extending Hoare logic's to include a treatment of time as in [Nei84] (where it is argued that this leads to a more natural formalisation of the informal approach). The significantly more difficult problem of analysing the *expected* (average) time-complexity has also been studied using Hoare-like logics *e.g.* [Ram79].

1.3 The Analysis of Functional Programs

Relatively little research into time-analysis has focused on the study of algorithms expressed as functional programs, although some attention has been given to the *mechanisation* of complexity analysis.

1.3.1 The Automatic Approaches

The first work in this area was that of Wegbreit [Weg75]. Wegbreit's METRIC system takes "simple" first-order pure LISP programs, and aims to automatically produce closed-form expressions for various aspects of time-complexity, expressed in terms of the size of the inputs to the program. The first step is the construction of recursive equations which describe the exact time-cost in terms of the inputs of the program. The system uses heuristics to determine a size-measure in which it will attempt to express complexity. This process derives (conditional) recurrence equations for which a solution is sought using a collection of standard recurrence techniques, together with some additional methods required to obtain average-case analyses. Direct extensions to these techniques to cope with a broader range of data-structures and algorithms are discussed in [ZZ89].

The ACE (Automatic Complexity Evaluator) system of Le Métayer [LeM85, LeM88b] analyses the worst-case behaviour of programs in FP ([Bac78]). The aim of the system is to derive a non-recursive closed-form FP expression (which can include functions like *length*) which describes an upper-bound on the running-time of a given function. The main idea here is that by performing the analysis at the FP level, the system can take advantage of the rich algebra of FP programs to perform much of the algebraic simplifications necessary. The method begins with the construction of a recursive function which computes the time-cost exactly, followed by simplifications using FP's axioms, together with an extendible rule-based transformation system which attempts to find non-recursive representations.

A similar functional perspective is taken by Rosendahl [Ros86, Ros89]. Here the language under analysis is a first-order pure lisp subset. As in Le Métayer's approach, the process begins with the construction of functions which take the same arguments as the original functions in the program and compute the time-cost. The main contribution of Rosendahl's work is to show how denotational-based analysis techniques related to abstract-interpretation can be used to construct a time-bound function tb from a step-counting function T, roughly satisfying $T \leq tb \circ S$, where S is some supplied size-measure. Further analyses and syntactic techniques are used to simplify the time-bound functions, leading ultimately to recurrence equations which can sometimes be solved using a library of a few standard methods.

Other work in this area focuses on the problems of average case analysis of simple recursive functions. This includes Flajolet's study of mathematical techniques necessary to solve the combinatorial problems that arise in the average-case analysis of certain classes of (recursive) algorithms [Fla85, FS81]. An account of the average-case analysis of FP programs is considered in [HC88]. A probabilistic semantics is used to express the expected complexity in terms of probability distributions. To handle structured data *attributed probabilistic grammars* are introduced.

The above works focus solely on the analysis of algorithms expressed as (pure) first-order recursion equations which are (implicitly) given a call-by-value operational semantics[1]. It could be argued that the aim of these works is *purely* the mechanisation of complexity analysis, and that the choice of a simple functional language is just a convenience:

- First-order recursion equations enable equations (*cost-functions*) describing exact time-cost to be constructed in a simple mechanical fashion.

- Cost-functions can be conveniently expressed in the original language, simplifying much of the necessary symbolic manipulation, and fall under the purview of techniques such as program transformation.

- Recursion equations lead more directly to the construction of mathematical recurrences, where there are well-studied methods for determining solutions (as well as automatic systems *e.g.* [CK77]).

Our interest is in the development of general techniques for reasoning about functional programs, rather than the choice of appropriate languages and methods for the mechanisation of complexity analysis. Since higher-order functions and lazy evaluation are essential elements of functional languages, we intend to consider the complexity issues that arise from their presence. There has been very little work which addresses these issues. Here we briefly review the work in this area, most of which will be considered in more detail within the relevant sections of the thesis.

[1]This means that arguments to a function are fully evaluated *before* the call

1.3.2 Analysing Higher-Order Functions

The problems involved in analysing the time-complexity of higher-order functions have been considered by Shultis [Shu85]. Shultis' study begins with the definition of a non-standard denotational semantics of a higher-order call-by-value functional language, which models both value and time-cost. This semantics provides a testing-ground for the development of an axiomatic theory for reasoning about time-cost. The theory is presented in the form of a logic in which properties about values and costs can be inferred. The key to this logic is the notion of the *toll* of an expression. The zeroth toll of an expression is just its evaluation cost. The first toll of an expression describes a function which gives the cost of an application of the expression. Tolls may be of an arbitrarily high level, reflecting the arbitrary levels of functional abstraction possible in the language.

A similar approach to the treatment of higher-order functions is considered by Le Métayer [LeM88a], where a range of (functional) program analysis problems are encoded in the language under analysis, and solved by program transformation techniques. In the time (and space) analysis of higher-order functions, associated with each function in the program we have a *family* of cost-functions. The first cost-function determines the cost of applying the function to one argument, the i^{th} (given i arguments) the cost of the i^{th} application.

The correctness of the above approaches are not considered in any detail (although Shultis provides a cost-model against which the logic could be validated). A general theory of Lisp-like computation which encompasses intensional properties has been considered by Talcott [Tal85b]. This provides a framework in which program calculi for reasoning about intensional properties of Lisp-like computations can be formalised.

1.3.3 Analysing Lazy Evaluation

In the analysis of programs under lazy evaluation, one of the main problems is the lack of compositionality: the cost due to an expression is not a simple composition of the costs of the sub-expressions. This is because, as the term "lazy" is intended to suggest, sub-expressions are not evaluated *more than necessary*, which implies that cost is context-dependent. In Bjerner's thesis [Bje89] a compositional theory for time analysis of the programs of Martin-Löf type-theory (a primitive-recursive functional language) is presented. The idea of compositional time-analysis is to parameterise the cost of computing an expression by a description of the amount of the result that is *needed* by the context in which it appears. For this Bjerner develops a calculus for reasoning about context based on the notion of "evaluation degrees".

A characterisation of "need" (more accurately "not-need") provided by a new form of strictness analysis [WH87] enabled Wadler to give a simpler account of Bjerner's approach [Wad88], applied to a (general) first-order functional language. The strictness-analysis angle of this approach also gives a natural notion of *approximation* of context-information, and gives rise to techniques for automatically reasoning about (approximate) contexts.

1.4 Overview of the Thesis

Our exploration of calculi for time analysis of functional programs focuses explicitly on the problems introduced by higher-order functions and lazy evaluation.

Underlying any treatment of time-complexity, either explicitly or implicitly, is some model of computation in which one is able to include descriptions of time-cost. In **Chapter 2** we introduce some of the basic ideas and techniques that will be used for developing and formalising calculi for time analysis. Using the example of a simple call-by-value first-order language, we define an abstract form of computational model via a structural operational semantics. Relative to this semantics, a suitable uniform "macro" measure of time is chosen, and for convenience this is integrated into the semantics to give the *step-counting semantics*. An externalisation of this semantic property by its encoding into the language itself provides a calculus for reasoning about time-cost. The time-cost for the evaluation of each of the functions in the program is computed by what we call *cost-functions*, which (following [LeM85]) are expressed in the original language. The cost functions are defined by a simple syntactic mapping over the function definitions, so for example a function like `sum` which sums the elements of a list:

```
sum(xs) = if null(xs) then 0 else hd(xs) + sum(tl(xs))
```

has the associated cost-function `csum` (which computes cost in terms of the number of recursive function calls)

```
csum(xs) = 1 + if null(xs) then 0 else csum(tl(xs))
```

which, using the usual proof methods for functional programs can be shown to be equal to `1 + length(xs)`. The correctness of the derivation step is easily shown against the step-counting semantics, and the approach forms the basic framework in which we consider the problems in a higher-order language.

In **Chapter 3** we develop techniques for reasoning about the time-cost of programs in an untyped call-by-value language with higher-order functions. The problem with constructing cost-functions for a language containing higher-order functions is that cost is dependent not only on the cost of computing the value of, but also on the cost of *applying* function-valued expressions. The first step towards the successful externalisation of time-cost via cost-functions is to modify the program so that function-valued expressions now come in the form of a (cost-function, function) pair. This method permits the construction of cost-functions, but only works for a restricted higher-order language. To extend the method to a general higher-order language we introduce structures called *cost-closures*. Cost-closures are (concrete) structures containing information about functions together with their corresponding cost-functions, and some essential arity information that is present in the underlying operational closures which implement the higher-order features. The methods developed are shown to be correct with respect to an appropriate operational model. Next we show how higher-order cost-functions can be factored in a way that reveals the cost-*structure* of higher-order functions in a more compositional manner. Finally in this chapter we show how an indirect application of these techniques can be applied to the analysis of call-by-name evaluation, via cost-preserving translations. Suitable classes of translations and the practicalities of this approach are discussed. A practical limitation is the cumbersome nature of the translated programs, which motivates a more direct approach.

In **Chapter 4** an approach to the time-analysis of a call-by-name language with "lazy" lists is developed. Instead of the construction of cost-functions (which is by no means straightforward), we consider a direct approach in which a suitably concise operational semantics forms the basis of simple equations for reasoning about time-cost. An advantage of this approach is its simplicity, but unlike the cost-function approaches we do not have the rich algebraic theories of functional programs at our disposal. In order to buy back some of the powers of equational reasoning we develop nonstandard notions of operational approximation and equivalence. A theory of *cost simulation*, a strong notion of operational approximation which includes time-cost, is developed by analogy with the notion of *(bi)simulation* in CCS [Mil89]. The central property of cost-simulation is that it is a precongruence relation, and this is proved.

Two drawbacks of the calculus in chapter 4 are firstly that it models call-by-name, when most non-strict languages are implemented using call-by-need, and secondly that it is not compositional. In **Chapter 5** we consider compositional calculus for reasoning about lazy (call-by-need) functional programs. In the first half of this chapter we develop a significant refinement of the approach in [Wad88]. Using context information in the form of a certain class of projections, we introduce two types of time-equation: *sufficient-time* equations which use information about "not-neededness" to give an upper-bound to the time for lazy evaluation, and *necessary-time* equations, a dual lower-bound, with better safety properties, which use information about "neededness" (strictness). The next step is the extension of these techniques to higher-order functions. In the absence of concise techniques for handling descriptions of "context" in the presence of higher-order functions[2] in the style of [WH87], we combine the first-order approach with a modified form of the cost-closure technique developed in chapter 3 to give an account of time-analysis in a higher-order lazy language.

Chapter 6 summarises the contributions of this thesis, and discusses directions for further work.

1.4.1 Some Notes to the Reader

This work goes some way towards allowing functional programmers to reason about intensional properties of their programs with something like the ease with which they currently reason about the extensional ones. There are a number of good introductions to functional programming, of which [BW88] and [FH88] are particularly recommended. Some background knowledge of functional-language semantics, both operational (in the structural style—see [Plo81, Kah87]), and denotational (see *e.g.* [Sch86]) are useful, but not essential.

The calculi we present are far from complete, not only in the sense implied by standard halting-problem arguments, but also in the range of methods that are considered. In general the mathematical techniques that are useful for the (average and worst-case) solution of equations describing time-cost are innumerable, and virtually none are considered here. However, the calculi reveal enough of the algorithmic structure of operationally opaque functional programs to permit the use of the more traditional techniques developed in the context of imperative programs—for an introduction the reader is referred to Aho, Hopcroft and Ullman's classic textbook

[2]although a recent generalisation of projections using partial equivalence relations [Hun90a] has opened up new possibilities in this area

[AHU74]. For an introduction to many pertinent mathematical methods see for example [GKP89].

Elements of chapters 3 and 5 have also been presented at *The 1989 Glasgow Functional Programming Workshop* [San89], and (in a shorter form) in the *Third European Symposium on Programming*, [San90]. Chapter 4 is largely self contained, as are the elements of Chapter 5 which refer to a first-order analysis, and these can be read independently from the rest of the thesis.

Chapter 2

Computational-Models and Cost-Functions

2.1 Introduction

In this chapter we begin our investigation of the time-analysis of functional programs by considering elements of the relatively well-studied analysis of a simple first-order language with a call-by-value semantics.

Our aim here is to give an introduction to some of the basic concepts explored in this thesis. We define a simple functional language and provide an abstract formulation of a cost-model, with which we can give a formal treatment of the idea of *cost-functions*. Our formulation of time-cost will be in terms of a structured operational semantics. This allows us to justify formally the derivation of cost-functions, which compute this time-cost property.

2.2 A First-Order Language

We begin with a simple first-order functional language, which will serve as an exemplar for the various simple functional languages introduced in this thesis. Expressions in this language are evaluated in the context of a set of mutually recursive function definitions of the form:

$$\begin{array}{rcl} f_1(x_1,\ldots,x_{n_1}) & = & e_1 \\ & \vdots & \\ f_k(x_1,\ldots,x_{n_k}) & = & e_k \end{array}$$

For simplicity we assume that there are distinct indexed function names, and that the formal parameters of the definitions are also indexed. This convention will simplify the following semantic and syntactic definitions, but will be dropped in the presentation of examples.

$$
\begin{array}{rcll}
e & ::= & f_i(e_1, \ldots, e_{n_i}) & \text{user-function applications} \\
 & | & p(e_1, \ldots, e_n) & \text{primitive-function applications} \\
 & | & \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 & \text{conditionals} \\
 & | & x & \text{identifiers} \\
 & | & c & \text{constants}
\end{array}
$$

Figure 2.1: First-order language Syntax

The syntax of expressions is given in figure 2.1. The exact set of primitive functions and constructors, ranged over by p, is unspecified here, although we will assume in our examples that we have a list constructor, and its associated `head` and `tail` operations, together with arithmetic and boolean functions.

2.3 Operational Semantics

2.3.1 Style

The meaning of terms in this language is given by an *operational* semantics, since it is clear that we will need some form of operational description of the language in order to reason formally about time-cost. In defining an operational semantics we do not follow the process of defining program execution (interpretation) via some specific abstract machine, but adopt the more abstract style of Plotkin's "Structural Operational Semantics". Our reason for adopting this approach is that it allows us to provide a semantics which contains sufficient detail to describe appropriate abstract measures of time-cost, but do not introduce more operational details than are necessary for this task. It is because the semantics must give *sufficient* operational details that we prefer this style of operational semantics to operational descriptions based on *rewriting* ideas [Klo80], since these approaches generally do not make sufficient commitment to evaluation-order, but require additional machinery, a *reduction strategy* , to capture these notions.

Generally, in the structural operational semantics approach, the semantics of a language is defined by a set of axioms and inductive rules. These define a logical system which allows us to infer semantic properties such as " expression e has type τ", or, "expression e evaluates in a single step to expression e'".

2.3.2 Typing Issues

Most modern functional languages are strongly typed, and are *polymorphic* in the style of ML [HMT88]. Whilst strong polymorphic types are important feature, we will not explicitly address typing issues for the simple languages presented in this thesis since strong typing is a *static* (compile-time) issue, and plays no direct role in the *dynamic* (run-time) behaviour of the language. Similarly, (parametric) polymorphism may be informally understood by the way in which each instance of a polymorphic function *behaves operationally in the same way.*

Recursive data-types do have relevance to complexity analysis, since they govern the classes of mathematical problems that arise (see, for example, [FS81]), but since this area is outside the scope of this thesis, for illustrative purposes we will simply assume a built-in list-type.

2.3.3 Dynamic Semantics: General Form

Here we give a general outline of the formulation of operational semantics in this thesis. Kahn *et al* [Kah87] focus on a particular style of structured operational semantics, which they call *Natural Semantics*, because of (superficial) similarities with natural deduction systems[1]. This work is concerned with characterising many aspects of programming language semantics, including type-systems and translation. We are only interested in the *dynamic* properties of languages, but the semantics we give is in the style of the dynamic elements of "natural semantics".

Sentences

The semantics for our language is defined as an inference system (a set of rules and axioms) which allows us to prove sentences of the form $\rho \vdash e \to v$. These sentences, or *judgements* are read as

Given environment ρ, expression e evaluates to value v

What we mean exactly by *values* and *environments* will be specific to the semantics defined.

[1]Semantics with a more genuine natural deduction flavour (*i.e.* the premises of a rule may be hypothetical) can be found in [BH87]

Semantics of this kind are sometimes referred to as *large step* semantics, because the judgements give us an "evaluation" relation between expressions and *values*. In a *small step* semantics the rules define smaller "reduction" steps. It is straightforward to convert a large-step semantics to a small-step version: see [Ast89] for examples. Our preference for the large-step style is because of its simplification of proofs of evaluation properties.

Rules and Axioms

The semantic definition is via an unordered collection of named rule schemas. Variables contained in a rule can be instantiated to give a specific instance of that rule (providing all instances of a variable are instantiated with the same term).

The *numerator* of a rule contains the premises; given proofs of (instances of) the premises we can construct a proof of (an instance of) the consequent (denominator). Rules containing no premises are called *axioms*.

In addition the rules may have side-conditions. These are applicability conditions not expressible as sentences.

2.3.4 A Dynamic Semantics

We use the above formalism to give a dynamic semantics for our language. The semantic rules are given in figure 2.2. Before we consider the properties of the semantics we give a brief explanation.

Environments

The environment to the left of the turnstile is used to map identifiers onto values. This environment is represented by a list, $\langle v_1, \ldots, v_n \rangle$ where the i^{th} element, v_i, is the value bound to identifier x_i. Rule **Id** illustrates the notation used: the value-environment, ρ is a list of values, where ρ_i denotes the i^{th} element.

Values

The domain of values includes the constants of the language, so that we have an axiom of the semantics that says that a constant evaluates to itself (rule **C**). (*NB* Since we will treat data type constructors as primitive functions, the set of values will also contain their concrete representations, but we do not give an explicit treatment here, other than for lists.)

Id $\rho \vdash_\phi x_i \rightarrow \rho_i$

C $\rho \vdash_\phi c \rightarrow c$

F $$\frac{\rho \vdash_\phi e_1 \rightarrow v_1 \;\cdots\; \rho \vdash_\phi e_{n_i} \rightarrow v_{n_i} \quad \langle v_1, \ldots, v_{n_i} \rangle \vdash_\phi e_i \rightarrow v}{\rho \vdash_\phi f_i(e_1, \ldots, e_{n_i}) \rightarrow v} \quad \textbf{if } \phi(f_i) = e_i$$

P $$\frac{\rho \vdash_\phi e_1 \rightarrow v_1 \;\cdots\; \rho \vdash_\phi e_n \rightarrow v_n}{\rho \vdash_\phi p(e_1, \ldots, e_n) \rightarrow v} \quad \textbf{if Apply}(p, \langle v_1, \ldots, v_n \rangle) = v$$

If.1 $$\frac{\rho \vdash_\phi e_1 \rightarrow \texttt{true} \quad \rho \vdash_\phi e_2 \rightarrow v}{\rho \vdash_\phi \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \rightarrow v}$$

If.2 $$\frac{\rho \vdash_\phi e_1 \rightarrow \texttt{false} \quad \rho \vdash_\phi e_3 \rightarrow v}{\rho \vdash_\phi \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \rightarrow v}$$

Figure 2.2: Dynamic Semantics

User-defined functions

We assume that we have a function-environment ϕ which maps function names to the body of that function, according to its definition. The judgements are parameterised by this environment, thus a sentence $\rho \vdash_\phi e \rightarrow v$ is read as

> Given environment ρ, in the context of definitions ϕ expression e evaluates to value v

We will not make the construction of this environment explicit; this can either be thought of as a separate semantic phase, or as part of rules which give meaning to *programs*, rather than expressions.

Primitive functions

The meaning of the primitive functions is given by a partial function **Apply**—further specification of this function is not given, although we assume that it is deterministic.

2.3.5 Basic Properties of the Semantics

The semantics specifies a call-by-value evaluation order—clearly illustrated in the rule for function application. There is a rule or axiom associated with each syntactic construct, with the exception of the (non-strict) conditional expression for which there are two rules whose applicability depends on the value of the condition.

To reason about time complexity using this semantics as a basis we rely on some basic properties of the system. An important property of the semantics is that it describes deterministic computation. An important (meta) property of the semantic system, from the point of view of its suitability as a cost-model, is that proofs are unique.

PROPOSITION **2.3.1** *For all ρ, ϕ, e, if Δ is a proof of $\rho \vdash_\phi e \rightarrow v$ for some v and Δ' is a proof of $\rho \vdash_\phi e \rightarrow v'$ for some v', then $\Delta = \Delta'$ and $v = v'$.*

PROOF Straightforward induction on the structure of Δ, by cases according to the last rule applied. $\square$

2.4 Defining Time Cost

In this thesis, for notational simplicity, but without loss of generality we choose to express time complexity as a macro-analysis. As a suitable "dominant" operation we will choose to express cost in terms of the number of non-primitive function calls, since this will always be sufficient to express asymptotic behaviour of programs, provided that we make the assumption that the primitive functions are simple "atomic" operations and hence do not "hide" computation.

2.4.1 Step-counting Semantics

In figure 2.3 we define an extended version of the semantics which has judgements of the form

$$\rho \vdash_\phi e \xrightarrow{s} \langle v, t \rangle$$

whose intended reading is

> *Given environment ρ, in the context of definitions ϕ, expression e evaluates to value v, involving t non-primitive function applications.*

If we think of the standard semantics as defining a partial function S from an environment and an expression to a value, then the corresponding step-counting function SS has the same domain (*i.e.* is as well-defined) and satisfies $S = \mathit{first} \circ SS$. The step-count component of the value is an externalisation of the number of instances of the rule **F** in the corresponding proof in the standard semantics. We can formalise this connection by defining $T(\Delta)$ to be the number of instances of rule **F** in proof Δ. T is defined inductively in the structure of the (labeled) proof (our convention will be to label instances of a rule on the right).

$$T\left(\frac{\Delta_1, \ldots, \Delta_k}{S}\;\mathbf{r}\right) = \begin{cases} 1 + T(\Delta_1) + \cdots + T(\Delta_k) & \text{if } \mathbf{r} = \mathbf{F} \\ T(\Delta_1) + \cdots + T(\Delta_k) & \text{otherwise} \end{cases}$$

$$T(S) = 0 \text{ if } S \text{ is an instance of an axiom}$$

Now we can state the correspondence as

PROPOSITION **2.4.1** *For all ρ, ϕ, e, v, t, we have*

$$\rho \vdash_\phi e \overset{s}{\rightarrow} \langle v, t\rangle$$

iff there exists a proof Δ of $\rho \vdash_\phi e \rightarrow v$ and $T(\Delta) = t$

Again it should be clear that the proof is a straightforward induction, and is safely omitted.

$$\mathbf{Id}_t \quad \rho \vdash_\phi x_i \overset{s}{\rightarrow} \langle \rho_i, 0\rangle$$

$$\mathbf{C}_t \quad \rho \vdash_\phi c \overset{s}{\rightarrow} \langle c, 0\rangle$$

$$\mathbf{F}_t \quad \frac{\begin{array}{c}\rho \vdash_\phi e_1 \overset{s}{\rightarrow} \langle v_1, t_1\rangle \;\cdots\; \rho \vdash_\phi e_{n_i} \overset{s}{\rightarrow} \langle v_{n_i}, t_{n_i}\rangle \\ \langle v_1, \ldots, v_{n_i}\rangle \vdash_\phi e_i \overset{s}{\rightarrow} \langle v, t\rangle\end{array}}{\rho \vdash_\phi f_i(e_1, \ldots, e_{n_i}) \overset{s}{\rightarrow} \langle v, 1 + t + t_1 + \cdots + t_{n_i}\rangle} \quad \textbf{if } \phi(f_i) = e_i$$

$$\mathbf{P}_t \quad \frac{\rho \vdash_\phi e_1 \overset{s}{\rightarrow} \langle v_1, t_1\rangle \;\cdots\; \rho \vdash_\phi e_n \overset{s}{\rightarrow} \langle v_n, t_n\rangle}{\rho \vdash_\phi p(e_1, \ldots, e_n) \overset{s}{\rightarrow} \langle v, t_1 + \cdots + t_n\rangle} \quad \textbf{if Apply}(p, \langle v_1, \ldots, v_n\rangle) = v$$

$$\mathbf{If}_t.1 \quad \frac{\rho \vdash_\phi e_1 \overset{s}{\rightarrow} \langle \texttt{true}, t_1\rangle \quad \rho \vdash_\phi e_2 \overset{s}{\rightarrow} \langle v, t_2\rangle}{\rho \vdash_\phi \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \overset{s}{\rightarrow} \langle v, t_1 + t_2\rangle}$$

$$\mathbf{If}_t.2 \quad \frac{\rho \vdash_\phi e_1 \overset{s}{\rightarrow} \langle \texttt{false}, t_1\rangle \quad \rho \vdash_\phi e_3 \overset{s}{\rightarrow} \langle v, t_3\rangle}{\rho \vdash_\phi \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \overset{s}{\rightarrow} \langle v, t_1 + t_3\rangle}$$

Figure 2.3: Step-Counting Semantics

2.5 Cost-Functions

The step-counting semantics can be viewed as a (specification of) a simple profiling interpreter. However, as a basis for reasoning about time-complexity it is a cumbersome tool, analogous to using the standard semantics as a basis for reasoning about extensional properties of functions. It would be more desirable to employ a calculus which would allow us to exploit "algebraic" and equational properties of the language, for example in the transformational style of [BD77], and the algebraic structure of the value domain. In this section we introduce *cost-functions.* These are functions corresponding to the functions in the program, which given the same argument values, compute the cost (corresponding to the step-count in the second component of the step-counting semantics).

For this language we can build such cost-functions as follows: For each equation of the form

$$f_i(x_1, \ldots, x_{n_i}) = e_i$$

we construct an equation which computes the cost (in terms of the number of non-primitive function calls) of applying f_i to a tuple of values. The cost equation is defined as:

$$cf_i(x_1, \ldots, x_{n_i}) = 1 + \mathcal{T}[\![e_i]\!]$$

where $\mathcal{T}$ is a syntax-directed mapping defined in figure 2.4.

$$\begin{array}{rcl}
\mathcal{T}[\![c]\!] & = & 0 \\
\mathcal{T}[\![x]\!] & = & 0 \\
\mathcal{T}[\![\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3]\!] & = & \mathcal{T}[\![e_1]\!] + \texttt{if } e_1 \texttt{ then } \mathcal{T}[\![e_2]\!] \texttt{ else } \mathcal{T}[\![e_3]\!] \\
\mathcal{T}[\![p\texttt{(}e_1, \ldots, e_n\texttt{)}]\!] & = & \mathcal{T}[\![e_1]\!] + \cdots + \mathcal{T}[\![e_n]\!] \\
\mathcal{T}[\![f_i\texttt{(}e_1, \ldots, e_n\texttt{)}]\!] & = & cf_i\texttt{(}e_1, \ldots, e_n\texttt{)} + \mathcal{T}[\![e_1]\!] + \cdots + \mathcal{T}[\![e_n]\!]
\end{array}$$

Figure 2.4: Cost-Function Construction Map

Syntax directed derivations of this form, for similar first order languages can be found in [Weg75, LeM85, Ros89]. These works focus on the techniques by which the recursive cost equations can be manipulated to achieve a non-recursive equation. Wegbreit refers to the first step (deriving cost-functions) as "local cost analysis", in which all the operations are counted using symbolic time-constants, while Rosendahl derives cost-functions which compute the number of primitive-function applications, which is referred to as "the step counting version" of the program.

Example

As a simple example of the above scheme, consider the list-append function defined as:

```
append(x,y)  =  if null(x) then y
                else cons(hd(x), append(tl(x),y))
```

From this definition, applying $\mathcal{T}$ we obtain the cost-function which computes the number of non-primitive function applications:

```
cappend(x,y)  =  1 + if null(x) then 0
                     else 0 + cappend(tl(x),y) + 0 + 0
```

From this it is simple to show that

```
cappend(x,y)  =  1 + length(x)
```

2.6 Correctness

The standard, and step-counting semantics provide the necessary machinery to reason about the correctness of cost-programs. The correctness is derived as follows:

DEFINITION **2.6.1** *If ϕ is some function environment which describes some function definitions*

$$f_i(x_1, \ldots, x_{n_i}) = e_i$$

then let ϕ' denote the environment which extends ϕ with the cost functions

$$cf_i(x_1, \ldots, x_{n_i}) = 1 + \mathcal{T}[\![e_i]\!]$$

□

THEOREM **2.6.2** *For all expressions e, and value environments ρ, and function-environments ϕ, if there exists a value v such that*

$$\rho \vdash_\phi e \xrightarrow{s} \langle v, t \rangle$$

for some t then it follows that

$$\rho \vdash_{\phi'} \mathcal{T}[\![e]\!] \rightarrow t$$

Note that this is not a total correctness ($\Leftrightarrow$)—it says nothing about non-termination or run-time errors in the evaluation of the original program. It is easy to see that non-termination will be inherited by the cost program, whereas run-time errors (*e.g.*`hd(nil)`) may not, and so the cost-program may be more defined than the original.

The proof follows by induction on the structure of the proof in the step-counting semantics, by cases according to the last rule applied.

PROOF We consider the case of function-application as an example. In this case the last rule must have the following form:

$$\boxed{\mathbf{F}_t \quad \frac{\rho \vdash_\phi e_1 \xrightarrow{s} \langle v_1, t_1\rangle \ \cdots\ \rho \vdash_\phi e_{n_i} \xrightarrow{s} \langle v_{n_i}, t_{n_i}\rangle \qquad \langle v_1,\ldots,v_{n_i}\rangle \vdash_\phi e_i \rightarrow \langle v, t\rangle}{\rho \vdash_\phi f_i(e_1,\ldots,e_{n_i}) \xrightarrow{s} \langle v, 1+t+t_1+\cdots+t_{n_i}\rangle}}$$

The inductive hypothesis is that the theorem holds for the antecedents of the rule, so we have

$$\rho \vdash_{\phi'} \mathcal{T}[\![e_j]\!] \rightarrow t_j \ , j = 1\ldots n_i \tag{2.1}$$

Now $\mathcal{T}[\![f_i(e_1,\ldots,e_{n_i})]\!] = cf_i(e_1,\ldots,e_{n_i}) + \mathcal{T}[\![e_1]\!] + \cdots + \mathcal{T}[\![e_{n_i}]\!]$ by definition, and $\phi'(cf_i) = 1 + \mathcal{T}[\![e_i]\!]$ by definition 2.6.1. The inductive hypothesis also gives us

$$\langle v_1,\ldots,v_{n_i}\rangle \vdash_{\phi'} \mathcal{T}[\![e_i]\!] \rightarrow t \tag{2.2}$$

and hence from the standard-semantics it is easily shown that

$$\langle v_1,\ldots,v_{n_i}\rangle \vdash_{\phi'} cf_i(e_1,\ldots,e_{n_i}) \rightarrow 1+t$$

which combined with 2.1 allows us to conclude (formally, via applications of rule **P** for the primitive function "+") that

$$\rho \vdash_{\phi'} \mathcal{T}[\![f_i(e_1,\ldots,e_{n_i})]\!] \rightarrow 1+t+t_1+\cdots+t_{n_i}$$

as required. □

2.7 Discussion

Our emphasis in this chapter has been on establishing an approach to time analysis by showing how cost-functions can be correctly constructed with respect to some appropriate *cost-model.* Since cost-functions in a first-order strict language are very straightforward and intuitive, they are usually derived (*e.g.* [LeM85, Weg75, ZZ89]) without formal justification. The importance of the explicit cost-model will be seen in the next chapter where we consider the less obvious treatment of a language with higher-order functions.

2.7.1 Denotational Descriptions

An alternative approach to providing a cost-model would be to present a non-standard denotational semantics describing the operational property of interest, but unrelated to the standard semantics (in the manner of, for example, Hudak's reference-count semantics [Hud87]), or via an "instrumented" semantics such as the step-counting semantics given in the later account of Rosendahl's work [Ros89]. Although formulating properties as least fixed point equations is appealing from the mathematical point of view, it is not entirely satisfactory since unless a formal connection to the actual operational property of interest is made, such a nonstandard denotational semantics is somewhat arbitrary. A possible justification in these cases is that the semantics is *obviously* correct with respect to the operational property of interest, but this is, at best, only convincing for first-order languages.

2.7.2 Other Operational Formalisms

There is much correspondence between the issues involved in the operational description of time-cost and elements of Talcott's intensional theory of Lisp-like computations [Tal85a]. In Talcott's terminology the step-counting function over proofs, T is an instance of a *derived property* (in general, analogous to properties of proof-trees), and the cost-function which computes it is a *derived program.* The operational presentation we have adopted is however based on the older and rather more familiar operational semantics style due to Plotkin, and is not so specialised to describing Lisp-like computation.

Chapter 3

Higher-Order Analysis

3.1 Introduction

In this chapter we develop a calculus for reasoning about a strict higher-order functional language. The calculus takes the form of that in the previous chapter: the main problem we solve is the construction of equations which compute the time-complexity of a given program. For the first-order, strict language this has been shown to be very intuitive and straightforward, and a few systems have been developed to mechanise the manipulation of the derived cost-functions. In this chapter we show how to derive cost-equations for a higher-order language. The derivation yields a functional program, is mechanisable, and thus provides a concise calculus for reasoning about the time-complexity of higher-order functions, and by further exploiting the functional nature of the equations could form the basis for an analysis tool for higher-order languages in the style of Rosendahl and Le Métayer's systems.

3.1.1 The Problem

To restate our formulation of the problem: given a program (which we will consider to be any expression, plus a set of mutually recursive functions) we seek to find a (syntax–directed) means of constructing a new program which computes the cost (in terms of the number of certain elementary operations) of executing any expression. Once again, without loss of generality we measure cost in terms of the number of non-primitive function applications.

We identify certain desirable criteria for such a cost-program:

- The cost program should be in a functional language — because we want to take advantage of the rich class of program transformation techniques and algebraic properties of functional programs.

- The cost program should be in the same language as the original program (although we may wish to consider a new semantic domain of *costs*) — this is desirable since it immediately gives an unambiguous semantics to cost-programs, and the existing technology can be used to execute them. (For example, the cost analysis of a strict first-order language, as presented in the previous chapter, does not require a language with higher-order functions). In addition, informal reasoning about cost can be tested against cost-function execution, and partially optimised cost-functions could provide a machine independent rapid profiling.

3.1.2 Overview

The solution to the problem of deriving cost-functions in a higher-order language is developed by first considering a restricted form of higher-order language in which the only functional values are named functions in the program (*i.e.* no lambda-expressions or curried functions). This motivates part of our solution, but the language is only "first-and-a-bit" order. This language is then extended to allow currying, (giving a fully general higher-order language) and the problems of deriving cost-functions are retackled.

The development of cost-functions presented in this chapter is via an informal motivation of the "translation schemes" for deriving cost-functions in a higher-order language. This is followed by a more rigorous study of correctness based on the operational semantics of the language.

We go on to show the pertinence of our methods by illustrating how a form of *factorisation* allows us to reason about higher-order cost-functions in a more compositional manner, based on an intuitive notion of inherent, or *context-free* cost in a higher-order function.

Finally we show how these techniques can be applied indirectly to the analysis of call-by-name evaluation, via cost-preserving *translations*. Classes of translations and the practicalities of this approach are considered.

3.2 A Restricted Higher Order Language

In this section we present a means of deriving cost programs for a simple higher order language. The cost program satisfies our criteria in that it is expressed in the same functional language as the original program.

Firstly we define our language. As before, a *program* consists of a set of recursive function definitions, together with a closed expression to be evaluated in the context of these definitions. The syntax of expressions in this language is defined in figure 3.1

$$
\begin{array}{rcll}
e & ::= & e(e_1, \ldots, e_j) & \text{(application)} \\
 & | & f_i & \text{(function)} \\
 & | & p & \text{(primitive function)} \\
 & | & x & \text{(identifier)} \\
 & | & c & \text{(constants)} \\
 & | & \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 & \text{(conditional)}
\end{array}
$$

Figure 3.1: Expression Syntax

We choose the same class of primitive functions p as in the previous chapter, namely, any strict (first-order) basic function (or constructor). Thus we have a restricted form of higher-order language with no currying—the language is not fully general because each expression that returns a function can only return one of the named functions in the program, or an uncurried primitive function.

To develop a method of analysing this language, consider the following simple example—an apply function defined as:

```
apply(f,x)  =  f(x)
```

The cost function associated with `apply` should have the form:

`Capply(f,x) = 1 +` *the cost of applying* `f` *to* `x`.

But how do we *syntactically* refer to the cost function associated with `f`? We do *not* wish to introduce a function which returns the cost function (at run time) associated with its argument, since this would not be supportable in a functional programming language (it would require a non-referentially transparent pointer-equality test).

Our solution (which is the key to the analysis) is to modify the original function definitions so that each *unapplied* user defined function is paired with its associated cost function. In addition, whenever an identifier is applied to its argument (a tuple), since the identifier must be bound to such a pair, the appropriate component of the pair must be applied to the tuple.

For example, given the simple program :

```
apply(f,x)  =  f(x)
inc(x)      =  x + 1
--------------------
apply(inc,x)
```

As an intermediate step, the program is modified to:

```
apply'(f,x)  =  fun(f)(x)
inc'(x)      =  x + 1
--------------------
apply((inc',Cinc),x)
```

The unapplied instance of `inc` is translated to the pair `(inc',Cinc)` and the function `inc'` is extracted by the new primitive selector function `fun`. We can now derive the complete cost-program:

```
apply'(f,x)  =  fun(f)(x)
inc'(x)      =  x + 1
Capply(f,x)  =  1 + cost(f)(x)
Cinc(x)      =  1
--------------------
Capply((inc',Cinc),x)
```

Generalising these ideas, the cost of evaluating any expression e is defined by the program scheme in figure 3.2.

The purpose of the function $\mathcal{V}$ is to translate *unapplied* user defined functions `f` into a pair `(f',cf)`, and, when a functional parameter is applied the appropriate

$$
\begin{array}{rcl}
f'_1(x_1 \ldots x_{n_1}) & = & \mathcal{V}[\![e_1]\!] \\
& \vdots & \\
f'_k(x_1 \ldots x_{n_k}) & = & \mathcal{V}[\![e_k]\!] \\
cf_1(x_1 \ldots x_{n_1}) & = & 1 + \mathcal{T} \circ \mathcal{V}[\![e_1]\!] \\
& \vdots & \\
cf_k(x_1 \ldots x_{n_k}) & = & 1 + \mathcal{T} \circ \mathcal{V}[\![e_k]\!] \\
\hline
\end{array}
$$

$$\mathcal{T} \circ \mathcal{V}[\![e]\!]$$

Figure 3.2: Simple Higher-Order Cost-Program Scheme

$$
\begin{array}{rcll}
\mathcal{V}[\![c]\!] & = & c & (3.1) \\
\mathcal{V}[\![x]\!] & = & x & (3.2) \\
\mathcal{V}[\![\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3]\!] & = & \texttt{if } \mathcal{V}[\![e_1]\!] \texttt{ then } \mathcal{V}[\![e_2]\!] \texttt{ else } \mathcal{V}[\![e_3]\!] & (3.3) \\
\mathcal{V}[\![p(e_1, \ldots, e_n)]\!] & = & p(\mathcal{V}[\![e_1]\!], \ldots, \mathcal{V}[\![e_n]\!]) & (3.4) \\
\mathcal{V}[\![f_i(e_1 \ldots e_n)]\!] & = & f'_i(\mathcal{V}[\![e_1]\!] \ldots \mathcal{V}[\![e_n]\!]) & (3.5) \\
\mathcal{V}[\![f_i]\!] & = & (f'_i, cf_i) & (3.6) \\
\mathcal{V}[\![p]\!] & = & (p, cp) & (3.7) \\
\mathcal{V}[\![e(e_1, \ldots, e_n)]\!] & = & \texttt{fun}(\mathcal{V}[\![e]\!])(\mathcal{V}[\![e_1]\!], \ldots, \mathcal{V}[\![e_n]\!]) & (3.8)
\end{array}
$$

Figure 3.3: Translation $\mathcal{V}$

component of this pair (*i.e.* `f'`) must be applied (extracted by the selector `fun`). $\mathcal{V}$ is a syntactic translation defined on the structure of expressions e in figure 3.3.

Rules (3.6) and (3.7) are relevant when functions are unapplied—they ensure that the functions are passed with their cost. Rule (3.8) ensures that the appropriate component of such a pair is applied when an expression evaluates to a function. Note that (overlapping) rules 3.4 and 3.5 are not strictly necessary, but give a small optimisation.

In figure 3.4 we define the cost-function constructor $\mathcal{T}$, over the syntax of the expressions $\mathcal{V}[\![e]\!]$.

$$
\begin{aligned}
\mathcal{T}[\![c]\!] &= 0 && (3.9)\\
\mathcal{T}[\![x]\!] &= 0 && (3.10)\\
\mathcal{T}[\![\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3]\!] &= \mathcal{T}[\![e_1]\!] + \texttt{if } e_1 \texttt{ then } \mathcal{T}[\![e_2]\!] \texttt{ else } \mathcal{T}[\![e_3]\!] && (3.11)\\
\mathcal{T}[\![p\texttt{(}e_1 \ldots e_n\texttt{)}]\!] &= \mathcal{T}[\![e_1]\!] + \cdots + \mathcal{T}[\![e_n]\!] && (3.12)\\
\mathcal{T}[\![f'_i\texttt{(}e_1 \ldots e_n\texttt{)}]\!] &= \mathcal{T}[\![e_1]\!] + \cdots + \mathcal{T}[\![e_n]\!] + cf_i\texttt{(}e_1 \ldots e_n\texttt{)} && (3.13)\\
\mathcal{T}[\![\texttt{fun(}e\texttt{)(}e_1 \ldots e_n\texttt{)}]\!] &= \mathcal{T}[\![e]\!] + \mathcal{T}[\![e_1]\!] + \cdots + \mathcal{T}[\![e_n]\!] \\
&\quad + \texttt{cost(}e\texttt{)(}e_1 \ldots e_n\texttt{)} && (3.14)\\
\mathcal{T}[\![(f'_i, cf'_i)]\!] &= 0 && (3.15)\\
\mathcal{T}[\![(p, cp)]\!] &= 0 && (3.16)
\end{aligned}
$$

Figure 3.4: Translation $\mathcal{T}$

Rules (3.9) to (3.13) are as in the first order analysis. In rule (3.14), the general application, we sum the cost of evaluating the function, the cost of evaluating each component of its argument tuple and the cost of the actual application. As before we take the costs associated with primitive function application to be zero.

3.2.1 Examples

Here is a simple example:

```
fmap(l,v)    =  if null(l) then nil
                else cons (hd(l)(v), fmap(tl(l),v))
inc(x)       =  x + 1
id(x)        =  x
incpair(x)   =  fmap(cons(id, cons(inc, nil)), x)

                     incpair(9)
```

The cost program derived from this is (with a few trivial simplifications):

```
fmap'(l,v)    =  if null(l) then nil
                 else cons (fun(hd(l))(v), fmap'(tl(l),v))
inc'(x)       =  x + 1
id'(x)        =  x
incpair'(x)   =  fmap'(cons((id',cid),cons((inc',cinc),nil)), x)
cfmap(l,v)    =  1 + if null(l) then 0
                     else cost(hd(l))(v) + cfmap(tl(l),v)
cinc(x)       =  1
cid(x)        =  1
cincpair(x)   =  1 +
                 cfmap(cons((id',cid),cons((inc',cinc),nil)), x)

                          cincpair(9)
```

The (non–primitive) reduction steps in the (informal[1]) evaluation of `incpair(9)`

[1]Rewriting expressions according to the defining equations using a (leftmost) innermost strategy. Evaluation will in fact be formalised in terms of a *relational*-style semantics, rather than term rewriting, however this presentation gives a good intuitive overview.

are :

```
incpair(9) -->
--> fmap(cons(id, cons(inc, nil)), 9)
--> cons (id(9), fmap(cons(inc, nil),9))
--> cons (9, fmap(cons(inc, nil),9))
--> cons (9, cons(inc(9), fmap(nil,9)))
--> cons (9, cons(10, fmap(nil,9)))
--> cons (9, cons(10, nil))
```

The evaluation of `Cincpair'` :

```
cincpair(9) -->
1 + cfmap(cons((id', cid), cons((inc', cinc), nil)), 9) -->
1 + 1 + cid(9) + cfmap(cons((inc',cinc), nil), 9) --> ...
--> 1 + 1 + 1 + 1 + cinc(9) + cfmap(nil, 9) --> ...
--> 6
```

as expected.

In the next section we extend the analysis to cope with curried functions, thus treating higher-order functions in their full generality.. The difficulty here is that no evaluation inside the body of a function occurs until the function is supplied with at least the number of arguments in its defining equation. The solution involves extending the pairs (f, cf) to include the *arity* of the function.

3.3 A Higher-Order Curried Language

3.3.1 The Problem

In this section we show how we can handle a fully curried language, so first we introduce such a language. The syntax of a program is shown in figure 3.5. Expressions syntax is defined in figure 3.6 (the two classes of expression, *exp* and *e* are simply used to disambiguate application). In the previous analysis, instances of *unapplied* user-defined functions were paired with the appropriate cost-function. In the context of *currying* we now have the possibility of *partially* applied functions:

DEFINITION **3.3.1** *The* arity *of a user-defined function is the number of arguments in the left-hand side of its defining equation.*

□

$$\begin{array}{|lcl|}
\hline
f_1\,x_1 \ldots x_{n_1} & = & exp_1 \\
 & \vdots & \\
f_k\,x_k \ldots x_{n_k} & = & exp_k \\
\hline
 & exp & \\
\hline
\end{array}$$

Figure 3.5: Higher-Order Program Scheme

$$\begin{array}{rcl}
exp & ::= & exp\ e \mid e \\
e & ::= & \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \mid (exp) \mid f_i \mid \\
 & & p_i \mid x \mid c
\end{array}$$

Figure 3.6: Higher-Order Expression Syntax

We shall refer to the arity of function f_i as n_i.

DEFINITION **3.3.2** *A user-defined function is* partially applied *when it is applied to fewer arguments than its* arity*.*

□

For reasons of efficiency and simplicity, there is no evaluation inside the body of a partially applied function—this is because such an evaluation strategy avoids (potentially expensive) name clashes, and is therefore a feature of most functional language implementations.

For each definition

$$f_i\,x_1 \ldots x_{n_i} = exp_i$$

we wish to construct a cost function

$$cf_i\,x_1 \ldots x_{n_i} = exp'_i$$

which computes the cost of applying f_i to n_i values. In the previous analysis, (syntactic) instances of unapplied functions were paired with their associated cost functions; we extend this for our curried language by also pairing *partially* applied functions.

This alone is not a sufficient extension. Consider the apply function:

```
apply f x  =  f x
```

We know that `f` will be bound to a fun/cost pair. In the non–curried language we also know that `f` has arity 1, and so this function is translated into:

```
apply' f x  =  fun f x
```

In a curried language `f` may have arity greater than one, and in this case we wish to construct a new fun/cost pair. This can be achieved by applying both components of the fun/cost pair (which is bound to `f`) to `x`. However, since we do not, in general, know the arity of the fun/cost pair that will be bound to `f`, we propose to *extend* the fun/cost pair to include some *arity* information.

3.3.2 Cost-closures

We extend the fun/cost pair by also including *arity* information. Thus an unapplied user-defined function f_i is translated into a triple (f_i, cf_i, n_i). This structure contains some of the information present *closure* which is constructed at run-time for partially applied functions[2]. We shall refer to this triple as a *cost-closure*.

Returning to our simple example, we introduce a selector function `arity` which returns the arity component of the cost–closure.

The translated `apply` function will now be:

```
apply' f x  =  if (arity f = 1) then (fun f x)
               else (fun f x, cost f x, arity f - 1)
```

i.e. if the arity of f is greater than one, a new cost–closure is built (just as, in the *execution* of the original definition, a new closure would be built). We can construct the corresponding cost function from this definition:

```
capply f' x = 1 + if (arity f' = 1) then (cost f' x)
                  else 0
```

[2]This arity-count is the size of the closure environment subtracted from the number of arguments in the function definition. In efficient implementations this is often represented explicitly, and for convenience will be present in the operational semantics given later.

The cost of building the closure is 0 since no evaluation takes place other than the construction of the closure, and (somewhat arbitrarily) we choose not to count this step.

We can use `apply` as the basis for our analysis, since this definition is in fact sufficient to deal with *any* application, since any application can be translated into an instance of `apply`: $exp\ e$ is equivalent to `apply` (*exp*) *e*. In order that we do not count this application of `apply` in our analysis we treat it as a primitive function, by removing the "`1 +` " in the definition of `capply'` above.

To aid presentation, we shall use `@` as the left–associative infix version of `apply'`, and similarly `c@` as the infix form of `capply'`. For example, `apply' (apply' f x) y` is equivalent to `f @ x @ y`. To summarise, functions `@` and `c@` satisfy:

$$(h, g, a)\ \texttt{@}\ e \ = \ \begin{cases} h\ e & \text{if } a = 1 \\ (h\ e, g\ e, a-1) & \text{otherwise} \end{cases}$$

$$(h, g, a)\ \texttt{c@}\ e \ = \ \begin{cases} g\ e & \text{if } a = 1 \\ 0 & \text{otherwise} \end{cases}$$

As before we shall define two syntax-directed translation maps $\mathcal{V}$ and $\mathcal{T}$.

- The purpose of $\mathcal{V}$ (figure 3.7) is to derive modified versions of the original functions such that all functions appear in the form of cost-closures, and all applications are via `@`.
- $\mathcal{T}$ (figure 3.8) defines the cost-expressions, using `c@` to extract the cost-component of the cost-closures.

The cost of evaluating any expression *exp* with respect to definitions

$$f_i\, x_1 \ldots x_{n_i} = exp_i \quad i = 1, \ldots, k$$

is then defined by the program given in figure 3.9.

The syntax of $\mathcal{V}$-translated expressions is:

$$exp' \ ::= \ exp'\ \texttt{@}\ e' \mid e'$$

$$\begin{aligned} e' \ ::= \ & \texttt{if}\ e'_1\ \texttt{then}\ e'_2\ \texttt{else}\ e'_3 \mid (exp') \mid (f'_i\ ,\ cf_i\ ,\ n_i\ ) \mid \\ & (p_i\ ,\ cp_i\ ,\ m_i\ ) \mid x \mid c \mid constant \end{aligned}$$

$\mathcal{T}$ is consequently defined over the syntax of expressions exp' and e' generated by $\mathcal{V}$, in figure 3.8.

$$
\begin{aligned}
\mathcal{V}[\![exp\ e]\!] &= \mathcal{V}[\![exp]\!]\ \texttt{@}\ \mathcal{V}[\![e]\!] && (3.17)\\
\mathcal{V}[\![\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3]\!] &= \texttt{if}\ \mathcal{V}[\![e_1]\!]\ \texttt{then}\ \mathcal{V}[\![e_2]\!]\ \texttt{else}\ \mathcal{V}[\![e_3]\!] && (3.18)\\
\mathcal{V}[\![\texttt{(}exp\texttt{)}]\!] &= \texttt{(}\mathcal{V}[\![exp]\!]\texttt{)} && (3.19)\\
\mathcal{V}[\![f_i]\!] &= \texttt{(}f'_i\ \texttt{,}\ cf_i\ \texttt{,}\ n_i\ \texttt{)} && (3.20)\\
\mathcal{V}[\![p_i]\!] &= \texttt{(}p_i\ \texttt{,}\ cp_i\ \texttt{,}\ m_i\ \texttt{)} && (3.21)\\
\mathcal{V}[\![c]\!] &= c && (3.22)\\
\mathcal{V}[\![x]\!] &= x && (3.23)
\end{aligned}
$$

Figure 3.7: Function Modification Map, $\mathcal{V}$

$$
\begin{aligned}
\mathcal{T}[\![exp'\ \texttt{@}\ e']\!] &= \mathcal{T}[\![exp']\!]\ \texttt{+}\ \mathcal{T}[\![e']\!]\ \texttt{+}\ \texttt{(}exp'\ \texttt{c@}\ e'\texttt{)}\\
\mathcal{T}[\![\texttt{if}\ e'_1\ \texttt{then}\ e'_2\ \texttt{else}\ e'_3]\!] &= \mathcal{T}[\![e'_1]\!]\ \texttt{+}\ \texttt{if}\ e'_1\ \texttt{then}\ \mathcal{T}[\![e'_2]\!]\ \texttt{else}\ \mathcal{T}[\![e'_3]\!]\\
\mathcal{T}[\![\texttt{(}exp'\texttt{)}]\!] &= \texttt{(}\mathcal{T}[\![exp']\!]\texttt{)}\\
\mathcal{T}[\![\texttt{(}p_i\ \texttt{,}\ cp_i\ \texttt{,}\ m_i\ \texttt{)}]\!] &= 0\\
\mathcal{T}[\![\texttt{(}f_i\ \texttt{,}\ cf_i\ \texttt{,}\ n_i\ \texttt{)}]\!] &= 0\\
\mathcal{T}[\![c]\!] &= \mathcal{T}[\![x]\!] \;=\; 0
\end{aligned}
$$

Figure 3.8: Cost-Expression Construction Map, $\mathcal{T}$

$$
\boxed{
\begin{array}{c}
\begin{aligned}
f'_1\ x_1 \ldots x_{n_1} &= \mathcal{V}[\![e_1]\!]\\
&\vdots\\
f'_k\ x_1 \ldots x_{n_k} &= \mathcal{V}[\![e_k]\!]\\[1ex]
cf_1\ x_1 \ldots x_{n_1} &= 1 + \mathcal{T} \circ \mathcal{V}[\![e_1]\!]\\
&\vdots\\
cf_k\ x_1 \ldots x_{n_k} &= 1 + \mathcal{T} \circ \mathcal{V}[\![e_k]\!]
\end{aligned}\\
\hline
\mathcal{T} \circ \mathcal{V}[\![e]\!]
\end{array}
}
$$

Figure 3.9: Higher-Order Cost-Program Scheme

3.3.3 Basic Optimisations

The code derived by the above translation schemes is rather more cumbersome than is necessary. This is because schemes introduce instances of `@` and `c@` for every application. Below we give some straightforward optimisations that simplify the cost-program considerably. It is worth noting that these simplifications can be achieved by building more elaborate translation schemes (as in the non-curried version) that identify the higher-order components of a program so as to introduce only the essential cost-closures. This approach would sacrifice the overall clarity of the method, and its subsequent proof.

The first simplification involves expressions of the form:

$$e'_0 \ \texttt{@} \ldots \texttt{@} \ e'_j$$

where the expression e'_0 is a syntactic instance of a cost-closure:

$$(\textit{function} \ , \ \textit{costfunction} \ , \ n \ )$$

When we have an expression of this form (*i.e.* we know n at compile-time) we can symbolically evaluate the `@`'s (and hence eliminate them). This is a trivial partial evaluation, and can be achieved by the following rules:

(i) If $j = n$ the above expression simplifies to:

$$\textit{function} \ e'_1 \ldots e'_j$$

This optimisation was included in the non-curried language, where $n = 1$ for all functions.

(ii) If $j > n$ the above expression simplifies to:

$$\textit{function} \ e'_1 \ldots e'_n \texttt{@} \ e'_{n+1} \texttt{@} \ \ldots \texttt{@} \ e'_j$$

This is simply an instance of the above case.

(iii) When $j < n$, the cost-closure is equivalent to

$$(\textit{function} \ e'_1 \ldots e'_j \ , \ \textit{costfunction} \ e'_1 \ldots e'_j \ , \ n - j \ )$$

although this is not necessarily an optimisation.

The second simplification (invoked after the above) involves expressions of the form:

$$(\mathit{function}\ ,\ \mathit{costfunction}\ ,\ n\ )\ \texttt{c@}\ e'$$

This is simplified (by evaluation of `c@`) to

$$\begin{cases} \mathit{costfunction}\ e' & \text{when } n = 1 \\ 0 & \text{otherwise} \end{cases}$$

Example

The following example illustrates the optimisation:

```
map f x  =  if (null x) then nil
            else (cons (f (hd x)) (map f (tl x)))
```

The step counting program derived from this is:

```
map' f x  =  if ((null,cnull,1) @ x) then nil
             else (cons,ccons,2) @ (f @ ((hd,chd 1) @ x))
                  @ ((map',cmap,2) @ f @ ((tl,ctl,1) @ x)))
cmap f x  =  1 + ((null,cnull,1) c@ x) +
             if ((null,cnull,1) @ x) then 0
             else (((cons,ccons,2) c@ (f @ ((hd,chd 1) @ x))
                  + ((cons,ccons,2) @ (f @ ((hd,chd 1) @ x))
                  c@ ((map',cmap,2) @ f @ ((tl,ctl,1) @ x)))
                  + f c@ ((hd,chd,1) @ x) + ((hd,chd,1) c@ x)
                  + ((map',cmap,2) @ f c@ ((tl,ctl,1) @ x)))
                  + ((map',cmap,2) c@ f) + ((tl,ctl,1) c@ x))
```

where we have simplified the body of the cost-function according to $exp + 0 = 0 + exp = exp$. Using the basic optimisations above, and removing the costs of applying primitive functions we get:

```
map' f x  =  if (null x) then nil
             else (cons (f @ (hd x)) (map' f (tl x)))
cmap f x  =  1 + if (null x) then 0
                 else (f c@ (hd x)) + (cmap f (tl x))
```

In the remainder of our examples we will assume these basic optimisations.

3.3.4 Further Example

Some Notation

For convenience we introduce a simple notation for the cost closures corresponding to the named functions in the program.

DEFINITION **3.3.3** *For all (literal) functions f (i.e. unapplied primitive or user-defined functions), let f^{cc} denote the cost-closure $\mathcal{V}[\![f]\!]$.*

□

The following example is chosen because it makes extensive use of currying: (It shows how curried higher-order functions can be used to represent lists):

```
CONS x y f  =  f x y
HEAD x      =  x hed
hed a b     =  a
TAIL x      =  x tal
tal a b     =  b
```

```
HEAD (CONS p q)
```

Evaluation of this expression yields, after three reductions:

```
HEAD (CONS p q)   -->
CONS p q hed      -->
hed p q           -->
p
```

The stepcounting program derived is as follows: (after simplification, and using the cost-closure notation)

```
CONS' x y f  =  f @ x @ y
HEAD' x      =  x @ hed^cc
hed' a b     =  a
TAIL' x      =  x @ tal^cc
tal' a b     =  b

cCONS x y f  =  1 + (f @ x c@ y) + (f c@ x)
cHEAD x      =  1 + (x c@ hed^cc)
ched a b     =  1
cTAIL x      =  1 + (x c@ tal^cc)
ctal a b     =  1
```

```
cHEAD (CONS^cc @ p @ q)
```

Evaluation of this expression yields the expected answer 3 (non-primitive reduction steps). Note that the $\mathcal{V}$ translation preserves the meaning of expressions (modulo: a partially applied function will evaluate to a cost-closure):

```
HEAD' (CONS^cc @ p @ q)                  --> ...
fun (CONS' p q, cCONS p q , 1) hed^cc    -->
CONS' p q hed^cc                         --> ...
fun (hd', chd,2) p q                     -->
hd' p q                               -->   p
```

In the next section we are required to formalise this property of $\mathcal{V}$ in the consideration of the overall correctness.

3.4 Correctness

The derived program computes the number of times a certain "step" is performed in the evaluation of the program. We formalise our intuitive model of "evaluation steps" via an *operational semantics* in the style of the previous chapter. We prove that the number of steps our derived program computes is correct with respect to the actual operational behaviour of the original program.

3.4.1 An Operational Semantics

The semantics is presented in much the same style as that in the previous chapter.

Environments

The environments used for binding identifiers to values, the *value-environment* (lists) will be written in the form:

$$\langle v_1, \ldots, v_n \rangle$$

and values are appended to the right of a value environment using operator "++".

DEFINITION **3.4.1** *The infix function ++ adds a value to the end of a list (environment):*

$$\langle v_1, \ldots, v_n \rangle {+\!\!+} v = \langle v_1, \ldots, v_n, v \rangle$$

□

If ρ is a list of values, then ρ_i denotes the i^{th} element.

We will assume that we have constructed a function-environment which maps the function-names to the right-hand-side of their definition. Informally we parameterise the turnstile in the sentences by this environment—it would be straightforward to include the construction of this environment in the semantic rules.

Values

The set of values will be the constants of the language, plus lists etc., together with *closures* to represent functional values. A closure is represented by a triple, written (f, n, ρ), where f is a function name (*i.e.* one of the f_i or p_i), n is a positive integer representing the arity of the *closure*, and ρ is the value-environment which binds zero or more of the formal parameters of f (from left to right). For any closure (f_i, k, ρ), the following property holds

$$\mid \rho \mid + k = n_i$$

where $\mid \rho \mid$ is the number of values in (*i.e.* the length of) ρ, n_i is the arity of the function f_i, and $1 \leq k \leq n_i$.

Figure 3.10 defines the standard dynamic operational semantics of the language. **App.1** is the case where application evaluates to a closure (*i.e.* a function). When a closure receives its *final* argument (**App.2/3**), the function can be evaluated in the context of the environment (with the final value appended onto the end). In the

App.1 $$\frac{\rho \vdash_\phi exp \rightarrow (f, n', \rho') \quad \rho \vdash_\phi e \rightarrow v'}{\rho \vdash_\phi exp\ e \rightarrow (f, n'-1, \rho' {+\!\!+} v')} \quad \textbf{if } n' > 1$$

App.2 $$\frac{\rho \vdash_\phi exp \rightarrow (f_i, 1, \rho') \quad \rho \vdash_\phi e \rightarrow v' \quad \rho' {+\!\!+} v' \vdash_\phi \phi(f_i) \rightarrow v}{\rho \vdash_\phi exp\ e \rightarrow v}$$

App.3 $$\frac{\rho \vdash_\phi exp \rightarrow (p_i, 1, \rho') \quad \rho \vdash_\phi e \rightarrow v'}{\rho \vdash_\phi exp\ e \rightarrow v} \quad \textbf{if Apply}(p_i, (\rho' {+\!\!+} v')) = v$$

Cond.1 $$\frac{\rho \vdash_\phi e_1 \rightarrow \texttt{true} \quad \rho \vdash_\phi e_2 \rightarrow v}{\rho \vdash_\phi \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \rightarrow v}$$

Cond.2 $$\frac{\rho \vdash_\phi e_1 \rightarrow \texttt{false} \quad \rho \vdash_\phi e_3 \rightarrow v}{\rho \vdash_\phi \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \rightarrow v}$$

Brac $$\frac{\rho \vdash_\phi exp \rightarrow v}{\rho \vdash_\phi \texttt{(}exp\texttt{)} \rightarrow v}$$

Userf $\rho \vdash_\phi f_i \rightarrow (f_i, n_i, \langle\rangle)$

Primf $\rho \vdash_\phi p_i \rightarrow (p_i, m_i, \langle\rangle)$ **if** $m_i = arity(p_i)$

Ident $\rho \vdash_\phi x_j \rightarrow \rho_j$

Const $\rho \vdash_\phi c \rightarrow c$

Figure 3.10: Dynamic Semantics

case of a user defined function (**App.2**), the body of the function is evaluated in the value-environment. We package the meaning of the primitive functions (**App.3**) using a semantic function **Apply**, which is assumed to be a deterministic partial function.

3.4.2 Step counting

The intuitive idea of an "evaluation step" corresponds to a rule application in the elaboration of an expression. The steps our cost program is intended to count correspond to rule **App.2** in the above semantics. Again we externalise this via a step-counting semantics whose judgements are of the form:

$$\rho \vdash_{\phi} exp \xrightarrow{s} \langle v, t \rangle$$

which is read as

> Given environment ρ and function environment ϕ, evaluation of expression *exp* yields value v, with t reductions of non-primitive function applications.

The step-counting semantics is defined in figure 3.11, and is constructed by extending the value component of the standard semantics with a count of the number of instances of rule **App.2**. We will not formalise the relationship between the standard and step-counting semantics any further, although this is straightforward (see proposition 2.4.1).

3.4.3 Correctness Criterion

The above semantics will enable us to state and prove some properties about the cost-program.

DEFINITION **3.4.2** *If ϕ is a function environment representing some definitions*

$$f_i\, x_1 \ldots x_{n_i} = exp_i$$

$i = 1, \ldots, k$*, then let* $\phi^{\mathcal{TV}}$ *denote the function environment representing functions*

$$\begin{array}{rcl} f_i'\, x_1 \ldots x_{n_i} & = & \mathcal{V}[\![exp_i]\!] \\ cf_i\, x_1 \ldots x_{n_i} & = & 1 + \mathcal{T} \circ \mathcal{V}[\![exp_i]\!] \end{array}$$

□

SApp.1
$$\frac{\rho \vdash_\phi exp \xrightarrow{s} \langle (f, n', \rho'), n_1 \rangle \quad \rho \vdash_\phi e \xrightarrow{s} \langle v', n_2 \rangle}{\rho \vdash_\phi exp\ e \xrightarrow{s} \langle (f, n'-1, \rho' {+\!\!+} v'), n_1 + n_2 \rangle} \quad \textbf{if } n' > 1$$

SApp.2
$$\frac{\begin{array}{l}\rho \vdash_\phi exp \xrightarrow{s} \langle (f_i, 1, \rho'), n_1 \rangle \\ \rho \vdash_\phi e \xrightarrow{s} \langle v', n_2 \rangle \\ \quad \rho' {+\!\!+} v' \vdash_\phi \phi(f_i) \xrightarrow{s} \langle v, n_3 \rangle\end{array}}{\rho \vdash_\phi exp\ e \xrightarrow{s} \langle v, n_1 + n_2 + n_3 + 1 \rangle}$$

SApp.3
$$\frac{\rho \vdash_\phi exp \xrightarrow{s} \langle (p_i, 1, \rho'), n_1 \rangle \quad \rho \vdash_\phi e \xrightarrow{s} \langle v', n_2 \rangle}{\rho \vdash_\phi exp\ e \xrightarrow{s} \langle v, n_1 + n_2 \rangle \qquad \textbf{if Apply}(p_i, (\rho' {+\!\!+} v')) = v}$$

SCond.1
$$\frac{\rho \vdash_\phi e_1 \xrightarrow{s} \langle \texttt{true}, n_1 \rangle \quad \rho \vdash_\phi e_2 \xrightarrow{s} \langle v, n_2 \rangle}{\rho \vdash_\phi \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \xrightarrow{s} \langle v, n_1 + n_2 \rangle}$$

SCond.2
$$\frac{\rho \vdash_\phi e_1 \xrightarrow{s} \langle \texttt{false}, n_1 \rangle \quad \rho \vdash_\phi e_3 \xrightarrow{s} \langle v, n_2 \rangle}{\rho \vdash_\phi \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \xrightarrow{s} \langle v, n_1 + n_2 \rangle}$$

SBrac
$$\frac{\rho \vdash_\phi exp \xrightarrow{s} \langle v, n \rangle}{\rho \vdash_\phi \texttt{(}exp\texttt{)} \xrightarrow{s} \langle v, n \rangle}$$

SUserf $\rho \vdash_\phi f_i \xrightarrow{s} \langle (f_i, n_i, \langle\rangle), 0 \rangle$

SPrimf $\rho \vdash_\phi p_i \xrightarrow{s} \langle (p_i, m_i, \langle\rangle), 0 \rangle \qquad \textbf{if } m_i = arity(p_i)$

SIdent $\rho \vdash_\phi x_j \xrightarrow{s} \langle \rho_j, 0 \rangle$

SConst $\rho \vdash_\phi c \xrightarrow{s} \langle c, 0 \rangle$

Figure 3.11: Step-Counting Semantics

The basic correctness criteria for cost-programs can then be stated as:

THEOREM **3.4.3 (Correctness Criterion)** *For all expressions exp, and function-environments ϕ, if there exists a value v such that*

$$\langle\rangle \vdash_{\phi} exp \xrightarrow{s} \langle v, n\rangle$$

for some n, then

$$\langle\rangle \vdash_{\phi^{\mathcal{TV}}} \mathcal{T} \circ \mathcal{V}[\![exp]\!] \rightarrow n$$

Note once more that this is not a total correctness ($\Longleftrightarrow$)—it says nothing about nontermination or run-time errors in the evaluation of the original program. It is easy to see that nontermination will be inherited by the cost program, whereas run-time errors (*e.g.* `hd(nil)`) may not, and so the cost-program may be more defined than the original, or may not terminate when the original program terminates with a run-time error.

Properties of $\mathcal{V}$

In order to prove the correctness of the cost-program scheme, we first need to consider properties of the cost-closure introduction map $\mathcal{V}$. The basic property that we need to show is that $\mathcal{V}$ preserves the "meaning" of expressions: since $\mathcal{V}$ translates functions into cost-closures, this will not be an equivalence. By a small abuse of notation, we define a mapping $(\cdot)^{cc}$ to relate values in the cost-program to values in the original program.

DEFINITION **3.4.4**

$$v^{cc} = \begin{cases} \langle v_1^{cc}, \cdots, v_k^{cc}\rangle & \textit{if } v = \langle v_1, \cdots, v_k\rangle \\ \langle (f', n, \rho^{cc}), (cf, n, \rho^{cc}), n\rangle & \textit{if } v = (f, n, \rho) \\ v & \textit{other values} \end{cases}$$

□

Thus closures are related to a three element list (the semantic representation of a cost-closure) in the cost program. (*NB* We are taking the semantics of a cost closure to be that of a three element list in order to show that we do not need an extended language to implement them). We extend this mapping to environments (*i.e.* a list of values) pointwise in the obvious way.

In addition, $\mathcal{V}$ introduces `@` as a means of cost-closure application. To avoid extending the semantics to deal with this operation (*i.e.* to show that we do not need to extend the semantics) we take the operational meaning of the term *exp'* `@` e' to be equivalent to that of the expression

```
if ((arity (exp')) = 1)                                            (3.24)
then (fun(exp') e')
else (fun(exp') e', cost(exp') e', (arity(exp')) - 1)
```

Since we are taking the semantics of the cost-closure to be that of a three element list (containing two *semantic* closures), `fun`, `cost` and `arity` have their obvious meanings in terms of list-primitives `hd` and `tl`.

The following lemma states that $\mathcal{V}$ preserves the meaning of programs, modulo ^{cc}.

LEMMA **3.4.5** *For all expressions exp, if there exists a value v such that*

$$\rho \vdash_{\phi} exp \rightarrow v$$

then it follows that

$$\rho^{cc} \vdash_{\phi^{T\mathcal{V}}} \mathcal{V}[\![exp]\!] \rightarrow v^{cc}$$

PROOF The proof is by induction on the structure of the proof of the judgement:

$$\rho \vdash_{\phi} exp \rightarrow v$$

The base cases correspond to use of the *axioms* (atomic derivations); the inductive cases correspond to use of each *rule* (by which a derivation is constructed from a set of smaller derivations). The inductive hypotheses for each case are that the lemma holds for the antecedents of the rule.

The proof of each inductive case is presented as a derivation tree (in the standard semantics), in which the leaves (subtrees) are the inductive hypotheses (or axioms). We abbreviate the evaluation of cost-closures, and their selector functions `fun`, `cost` and `arity` to simplify presentation.

$$\boxed{\textbf{Userf} \quad \rho \vdash_{\phi} f_i \rightarrow (f_i, n_i, \langle\rangle)}$$

From the definition of $\mathcal{V}$, $\mathcal{V}[\![f_i]\!] = (f_i'\ ,\ cf_i\ ,\ n_i\ )$. Cost closures are treated as three element lists, so we have

$$\rho^{cc} \vdash_{\phi^{T\mathcal{V}}} \mathcal{V}[\![f_i]\!] \rightarrow \langle (f_i', n_i, \langle\rangle), (cf_i, n_i, \langle\rangle), n_i \rangle$$

Now by definition of ^{cc},

$$(f_i, n_i, \langle\rangle)^{cc} = \langle (f'_i, n_i, \langle\rangle), (cf_i, n_i, \langle\rangle), n_i \rangle$$

and so $\rho^{cc} \vdash_{\phi^{T\mathcal{V}}} \mathcal{V}[\![f_i]\!] \to (f_i, n_i, \langle\rangle)^{cc}$ as required.

$\boxed{\textbf{Primf}}$

Similar to above.

$$\boxed{\textbf{Ident} \quad \rho \vdash_\phi x_j \to \rho_j}$$

Now $\mathcal{V}[\![x_j]\!] = x_j$ by definition, and so by definition of ^{cc}

$$\rho^{cc} \vdash_{\phi^{T\mathcal{V}}} \mathcal{V}[\![x_j]\!] \to \rho^{cc}{}_j \; = (\rho_j)^{cc}$$

$\boxed{\textbf{Const}, \textbf{Brac}}$

Straightforward.

$$\boxed{\textbf{App.1} \quad \frac{\rho \vdash_\phi exp \to (f, n', \rho') \quad \rho \vdash_\phi e \to v'}{\rho \vdash_\phi exp\ e \to (f, n'-1, \rho' {+\!\!+} v')} \quad \textbf{if } n' > 1}$$

Using the fact that

$$(f, n', \rho')^{cc} = \langle (f', n', \rho'^{cc}), (cf, n', \rho'^{cc}), n' \rangle$$

the inductive hypothesis gives:

$$\rho^{cc} \vdash_{\phi^{T\mathcal{V}}} \mathcal{V}[\![exp]\!] \to \langle (f', n', \rho'^{cc}), (cf, n', \rho'^{cc}), n' \rangle \tag{3.25}$$

where $n' > 1$, and

$$\rho^{cc} \vdash_{\phi^{T\mathcal{V}}} \mathcal{V}[\![e]\!] \to v'^{cc} \tag{3.26}$$

We must now show that

$$\rho^{cc} \vdash_{\phi^{T\mathcal{V}}} \mathcal{V}[\![exp\ e]\!] \to (f, n'-1, \rho' {+\!\!+} v')^{cc}$$

By definition we have

$$\mathcal{V}[\![exp\ e]\!] = \mathcal{V}[\![exp]\!]\ \texttt{@}\ \mathcal{V}[\![e]\!]$$

and

$$\begin{aligned}(f, n'-1, \rho' +\!\!+ v')^{cc} = \ & \langle (f', n'-1, (\rho' +\!\!+ v')^{cc}),\\ & (cf, n'-1, (\rho' +\!\!+ v')^{cc}),\\ & n'-1\rangle\end{aligned}$$

From the inductive hypotheses we can derive

$$\frac{\dfrac{\begin{array}{c}(3.25)\\ \vdots\ \ \texttt{fun}\text{ defn}\end{array}}{\rho^{cc} \vdash_{\phi^{T\mathcal{V}}} \texttt{fun}\ \mathcal{V}[\![exp]\!] \rightarrow (f', n', \rho'^{cc})} \quad (3.26)}{\rho^{cc} \vdash_{\phi^{T\mathcal{V}}} \texttt{fun}\ \mathcal{V}[\![exp]\!]\mathcal{V}[\![e]\!] \rightarrow (f', n'-1, \rho'^{cc} +\!\!+ v'^{cc})}\ \textbf{App.1} \tag{3.27}$$

where the function `fun` simply extracts the first component of the cost-closure—the inferences have been omitted. Similarly we can derive

$$\frac{\dfrac{\begin{array}{c}(3.25)\\ \vdots\ \ \texttt{cost}\text{ defn.}\end{array}}{\rho^{cc} \vdash_{\phi^{T\mathcal{V}}} \texttt{cost}\ \mathcal{V}[\![exp]\!] \rightarrow (cf, n', \rho'^{cc})} \quad (3.26)}{\rho^{cc} \vdash_{\phi^{T\mathcal{V}}} \texttt{cost}\ \mathcal{V}[\![exp]\!]\mathcal{V}[\![e]\!] \rightarrow (cf, n'-1, \rho'^{cc} +\!\!+ v'^{cc})}\ \textbf{App.1} \tag{3.28}$$

and

$$\begin{array}{c}(3.25)\\ \vdots\ \ \texttt{arity}\text{ defn.}\\ \rho^{cc} \vdash_{\phi^{T\mathcal{V}}} \texttt{arity}\ \mathcal{V}[\![exp]\!] \rightarrow n'\end{array} \tag{3.29}$$

Applying the primitive function rules, we also have

$$\rho^{cc} \vdash_{\phi^{T\mathcal{V}}} \texttt{(arity}\ \mathcal{V}[\![exp]\!]\texttt{) - 1} \rightarrow n'-1 \tag{3.30}$$

Putting 3.27, 3.28 and 3.30 together, (with the assumption that cost-closures are implemented as lists) we have

$$\begin{aligned}\rho^{cc} \vdash_{\phi^{T\mathcal{V}}} & \texttt{(fun(}\mathcal{V}[\![exp']\!]\texttt{)}\ e'\texttt{, cost(}\mathcal{V}[\![exp']\!]\texttt{)}\ \mathcal{V}[\![\texttt{e'}]\!]\texttt{, (arity(}\mathcal{V}[\![exp']\!]\texttt{)) - 1)} \rightarrow\\ & \langle (f', n'-1, \rho'^{cc} +\!\!+ v'^{cc}), (cf, n'-1, \rho'^{cc} +\!\!+ v'^{cc}), n'-1\rangle\end{aligned} \tag{3.31}$$

Since $n' > 1$ then we expect that

$$\rho^{cc} \vdash_{\phi^{T\mathcal{V}}} \texttt{(eq (arity (}\mathcal{V}[\![exp]\!]\texttt{)) 1)} \rightarrow \texttt{false} \tag{3.32}$$

Now we have

$$\frac{(3.32) \quad (3.31)}{\begin{aligned}\rho^{cc} \vdash_{\phi^{T\mathcal{V}}} \mathcal{V}[\![exp]\!]\texttt{@}\mathcal{V}[\![e]\!] \rightarrow \ & \langle (f', n'-1, \rho'^{cc} +\!\!+ v'^{cc}),\\ & (cf, n'-1, \rho'^{cc} +\!\!+ v'^{cc}),\\ & n'-1\rangle\end{aligned}}\ \textbf{Cond.2} \tag{3.33}$$

and the required result follows from the equivalence:

$$\rho'^{cc} +\!\!+ v'^{cc} = (\rho' +\!\!+ v')^{cc}$$

$$\boxed{\textbf{App.2} \quad \frac{\rho \vdash_\phi exp \rightarrow (f_i, 1, \rho') \quad \rho \vdash_\phi e \rightarrow v' \quad \rho' +\!\!+ v' \vdash_\phi \phi(f_i) \rightarrow v}{\rho \vdash_\phi exp\ e \rightarrow v}}$$

In this case the inductive hypothesis gives:

$$\rho^{cc} \vdash_{\phi^{\mathcal{TV}}} \mathcal{V}[\![exp]\!] \rightarrow \langle (f'_i, 1, \rho'^{cc}), (cf_i, 1, \rho'^{cc}), 1 \rangle \tag{3.34}$$

$$\rho^{cc} \vdash_{\phi^{\mathcal{TV}}} \mathcal{V}[\![e]\!] \rightarrow v'^{cc} \tag{3.35}$$

$$(\rho' +\!\!+ v')^{cc} \vdash_{\phi^{\mathcal{TV}}} \mathcal{V}[\![\phi(f_i)]\!] \rightarrow v^{cc} \tag{3.36}$$

From the definitions of $\phi^{\mathcal{TV}}$ we have

$$\phi(f_i) = e_i \Longrightarrow \phi^{\mathcal{TV}}(f'_i) = \mathcal{V}[\![e_i]\!]$$

So (3.36) is equivalent to the sentence

$$\rho'^{cc} +\!\!+ v'^{cc} \vdash_{\phi^{\mathcal{TV}}} \phi^{\mathcal{TV}}(f'_i) \rightarrow v^{cc} \tag{3.37}$$

We can also infer

$$\begin{array}{c} (3.34) \\ \vdots \quad \texttt{fun}\text{ defn} \\ \rho^{cc} \vdash_{\phi^{\mathcal{TV}}} \texttt{fun}\ \mathcal{V}[\![exp]\!] \rightarrow (f', 1, \rho'^{cc}) \end{array} \tag{3.38}$$

$$\begin{array}{c} (3.34) \\ \vdots \quad \texttt{arity}\text{ defn.} \\ \rho^{cc} \vdash_{\phi^{\mathcal{TV}}} \texttt{arity}\ \mathcal{V}[\![exp]\!] \rightarrow 1 \end{array} \tag{3.39}$$

and in this case we also have

$$\rho^{cc} \vdash_{\phi^{\mathcal{TV}}} \texttt{(eq (arity (}\mathcal{V}[\![exp]\!]\texttt{)) 1)} \rightarrow \texttt{true}\,. \tag{3.40}$$

Finally we can infer

$$\dfrac{(3.40) \qquad \dfrac{(3.38) \quad (3.35) \quad (3.37)}{\rho^{cc} \vdash_{\phi^{\mathcal{TV}}} \texttt{fun}\mathcal{V}[\![exp]\!]\mathcal{V}[\![e]\!] \rightarrow v^{cc}}\ \textbf{App.2}}{\rho^{cc} \vdash_{\phi^{\mathcal{TV}}} \mathcal{V}[\![exp]\!]\texttt{@}\mathcal{V}[\![e]\!] \rightarrow v^{cc}}\ \textbf{Cond.1}$$

as required.

App.3

Similar to previous case except that we must make the following restriction on primitive functions:

$$\textbf{Apply}(p_i, \rho) = v \Longrightarrow \textbf{Apply}(p_i, \rho^{cc}) = v^{cc}$$

for any environment ρ. This class of primitive functions includes any first-order function (since first-order objects are unchanged by $\mathcal{V}$) (*e.g.* `sub`,`eq`) and potentially higher-order ones like `head`, as well as `cons`. Primitive functions outside this category would need a specific translation in $\mathcal{V}$.

$$\textbf{Cond.1} \quad \frac{\rho \vdash_\phi e_1 \rightarrow \texttt{true} \quad \rho \vdash_\phi e_2 \rightarrow v}{\rho \vdash_\phi \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \rightarrow v}$$

The induction hypothesis gives:

$$\rho^{cc} \vdash_{\phi^{\mathcal{TV}}} \mathcal{V}[\![e_1]\!] \rightarrow \texttt{true}^{cc} \tag{3.41}$$

$$\rho^{cc} \vdash_{\phi^{\mathcal{TV}}} \mathcal{V}[\![e_2]\!] \rightarrow v^{cc} \tag{3.42}$$

Now since $\texttt{true}^{cc} = \texttt{true}$ (from the definition of ^{cc}), we can derive

$$\frac{(3.41) \quad (3.42)}{\rho^{cc} \vdash \texttt{if } \mathcal{V}[\![e_1]\!] \texttt{ then } \mathcal{V}[\![e_2]\!] \texttt{ else } \mathcal{V}[\![e_3]\!] \rightarrow v^{cc}} \ \textbf{Cond.1}$$

as required

Cond.2

Similar to the previous case. □

Properties of $\mathcal{T}$

The correctness criterion is an immediate corollary of the theorem below. The meaning of the cost-apply function (exp' `c@` e') is taken to be that of the expression

```
if (eq (arity (exp')) 1) then (cost (exp') e')          (3.43)
else 0
```

In addition we define the cost-functions associated with primitive functions to be zero, *i.e.* $\phi^{\mathcal{TV}}(cp_i) = 0$.

THEOREM **3.4.6** *For all expressions exp, function environments ϕ, and value environments ρ, if there exists a value v such that*

$$\rho \vdash_\phi exp \xrightarrow{s} \langle v, t\rangle$$

for some t, then

$$\rho^{cc} \vdash_{\phi^{\mathcal{T}\mathcal{V}}} \mathcal{T} \circ \mathcal{V}[\![exp]\!] \rightarrow t$$

PROOF Proof by induction on the structure of the derivation of

$$\rho \vdash_\phi exp \xrightarrow{s} \langle v, t\rangle$$

There is a case for each rule in the semantics.

$$\boxed{\textbf{SUserf} \quad \rho \vdash_\phi f_i \xrightarrow{s} \langle (f_i, n_i, \langle\rangle), 0\rangle}$$

$\mathcal{T} \circ \mathcal{V}[\![f_i]\!] = 0$ by the definition of $\mathcal{T}$ and $\mathcal{V}$. Therefore $\rho^{cc} \vdash_{\phi^{\mathcal{T}\mathcal{V}}} \mathcal{T} \circ \mathcal{V}[\![f_i]\!] \rightarrow 0$.

$$\boxed{\textbf{SPrimf}, \textbf{ SIdent}, \textbf{ SConst}, \textbf{ SBrac}}$$

Similar to above.

$$\boxed{\textbf{SApp.1} \quad \frac{\rho \vdash_\phi exp \xrightarrow{s} \langle (f, n', \rho'), n_1\rangle \quad \rho \vdash_\phi e \xrightarrow{s} \langle v', n_2\rangle}{\rho \vdash_\phi exp\ e \xrightarrow{s} \langle (f, n'-1, \rho' +\!\!+ v'), n_1 + n_2\rangle} \quad \textbf{if } n' > 1}$$

The induction hypothesis gives us

$$\rho^{cc} \vdash_{\phi^{\mathcal{T}\mathcal{V}}} \mathcal{T} \circ \mathcal{V}[\![exp]\!] \rightarrow n_1 \tag{3.44}$$

$$\rho^{cc} \vdash_{\phi^{\mathcal{T}\mathcal{V}}} \mathcal{T} \circ \mathcal{V}[\![e]\!] \rightarrow n_2 \tag{3.45}$$

Now from the definition of $\mathcal{T}$and $\mathcal{V}$ we have:

$$\mathcal{T} \circ \mathcal{V}[\![exp\ e]\!] = \mathcal{T} \circ \mathcal{V}[\![exp]\!] + \mathcal{T} \circ \mathcal{V}[\![e]\!] + \mathcal{V}[\![exp]\!] \ \texttt{c@}\ \mathcal{V}[\![e]\!] \tag{3.46}$$

The theorem requires us to show

$$\rho^{cc} \vdash_{\phi^{\mathcal{T}\mathcal{V}}} \mathcal{T} \circ \mathcal{V}[\![exp\ e]\!] \rightarrow n_1 + n_2$$

From the inductive hypotheses it will be sufficient to show that

$$\rho^{cc} \vdash_{\phi^{\mathcal{TV}}} \mathcal{V}[\![exp]\!] \;\texttt{c@}\; \mathcal{V}[\![e]\!] \rightarrow 0$$

From lemma 3.4.5, and from the fact that

$$\rho \vdash_\phi exp \xrightarrow{s} \langle (f, n', \rho'), n_1 \rangle \;\Rightarrow\; \rho \vdash_\phi exp \rightarrow (f, n', \rho')$$

it follows that:

$$\rho^{cc} \vdash_{\phi^{\mathcal{TV}}} \mathcal{V}[\![exp]\!] \rightarrow \langle (f', n', \rho'^{cc}), (cf, n', \rho'^{cc}), n' \rangle \tag{3.47}$$

where $n' > 1$, and so

$$\rho^{cc} \vdash_{\phi^{\mathcal{TV}}} \mathcal{V}[\![exp]\!] \;\texttt{c@}\; \mathcal{V}[\![e]\!] \rightarrow 0$$

(by evaluation of `c@` —straightforward derivation omitted), as required.

$$\boxed{\textbf{SApp.2} \quad \frac{\rho \vdash_\phi exp \xrightarrow{s} \langle (f_i, 1, \rho'), n_1 \rangle \quad \rho \vdash_\phi e \xrightarrow{s} \langle v', n_2 \rangle \quad \rho' {+\!\!+} v' \vdash_\phi \phi(f_i) \xrightarrow{s} \langle v, n_3 \rangle}{\rho \vdash_\phi exp\ e \xrightarrow{s} \langle v, n_1 + n_2 + n_3 + 1 \rangle}}$$

The induction hypothesis gives us

$$\rho^{cc} \vdash_{\phi^{\mathcal{TV}}} \mathcal{T} \circ \mathcal{V}[\![exp]\!] \rightarrow n_1 \tag{3.48}$$

$$\rho^{cc} \vdash_{\phi^{\mathcal{TV}}} \mathcal{T} \circ \mathcal{V}[\![e]\!] \rightarrow n_2 \tag{3.49}$$

$$(\rho' {+\!\!+} v')^{cc} \vdash_{\phi^{\mathcal{TV}}} \mathcal{T} \circ \mathcal{V}[\![\phi(f_i)]\!] \rightarrow n_3 \tag{3.50}$$

The theorem requires us to show

$$\rho^{cc} \vdash_{\phi^{\mathcal{TV}}} \mathcal{T} \circ \mathcal{V}[\![exp\ e]\!] \rightarrow n_1 + n_2 + n_3 + 1$$

From the inductive hypotheses and (3.46) it will be sufficient to show that

$$\rho^{cc} \vdash_{\phi^{\mathcal{TV}}} \mathcal{V}[\![exp]\!] \;\texttt{c@}\; \mathcal{V}[\![e]\!] \rightarrow n_3 + 1$$

From lemma 3.4.5, in this case we have:

$$\rho^{cc} \vdash_{\phi^{\mathcal{TV}}} \mathcal{V}[\![exp]\!] \rightarrow \langle (f'_i, 1, \rho'^{cc}), (cf_i, 1, \rho'^{cc}), 1 \rangle \tag{3.51}$$

and

$$\rho^{cc} \vdash_{\phi^{\mathcal{TV}}} \mathcal{V}[\![e]\!] \rightarrow v'^{cc} \tag{3.52}$$

From the definitions of ϕ and ^{cc} we have $\phi(f_i) = e_i$ for some expression e_i, and that

$$(\rho' \mathbin{++} v')^{cc} = \rho'^{cc} \mathbin{++} v'^{cc}$$

So we can restate (3.50) as the sentence

$$\rho'^{cc} \mathbin{++} v'^{cc} \vdash_{\phi^{\mathcal{TV}}} \mathcal{T} \circ \mathcal{V}[\![e_i]\!] \to n_3 \tag{3.53}$$

From (3.51) we can infer

$$\rho^{cc} \vdash_{\phi^{\mathcal{TV}}} \texttt{(eq (arity (}\mathcal{V}[\![exp]\!]\texttt{)) 1)} \to \texttt{true} \tag{3.54}$$

$$\rho^{cc} \vdash_{\phi^{\mathcal{TV}}} \texttt{cost}\ (\mathcal{V}[\![exp]\!]) \to (cf_i, 1, \rho'^{cc}) \tag{3.55}$$

Now $\phi^{\mathcal{TV}}(cf_i) = 1 + \mathcal{T} \circ \mathcal{V}[\![e_i]\!]$ by definition of the cost program, and so from 3.53 we can infer

$$\rho'^{cc} \mathbin{++} v'^{cc} \vdash_{\phi^{\mathcal{TV}}} \phi^{\mathcal{TV}}(cf_i) \to 1 + n_3 \tag{3.56}$$

We can now derive

$$\cfrac{(3.54) \qquad \cfrac{(3.55)\quad (3.52)\quad (3.56)}{\rho^{cc} \vdash_{\phi^{\mathcal{TV}}} \texttt{cost}(\mathcal{V}[\![exp]\!])\mathcal{V}[\![e]\!] \to 1 + n_3}\ \textbf{App.2}}{\rho^{cc} \vdash_{\phi^{\mathcal{TV}}} \mathcal{V}[\![exp]\!]\ \texttt{c@}\ \mathcal{V}[\![e]\!] \to 1 + n_3}\ \textbf{Cond.1}$$

and so

$$\rho^{cc} \vdash_{\phi^{\mathcal{TV}}} \mathcal{T} \circ \mathcal{V}[\![exp\ e]\!] \to 1 + n_1 + n_2 + n_3$$

as required

SApp.3

Similar to previous case, taking cost-functions for all primitive functions to be zero.

$$\textbf{SCond.1} \quad \frac{\rho \vdash_\phi e_1 \xrightarrow{s} \langle \texttt{true}, n_1 \rangle \quad \rho \vdash_\phi e_2 \xrightarrow{s} \langle v, n_2 \rangle}{\rho \vdash_\phi \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \xrightarrow{s} \langle v, n_1 + n_2 \rangle}$$

The induction hypothesis gives us

$$\rho^{cc} \vdash_{\phi^{\mathcal{TV}}} \mathcal{T} \circ \mathcal{V}[\![e_1]\!] \to n_1 \tag{3.57}$$

$$\rho^{cc} \vdash_{\phi^{\mathcal{TV}}} \mathcal{T} \circ \mathcal{V}[\![e_2]\!] \to n_2 \tag{3.58}$$

By definition of $\mathcal{T}$ and $\mathcal{V}$ we have:

$$\begin{aligned}&\mathcal{T}\circ\mathcal{V}[\![\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3]\!] =\\ &\qquad \mathcal{T}\circ\mathcal{V}[\![e_1]\!] + \texttt{if}\ \mathcal{V}[\![e_1]\!]\ \texttt{then}\ \mathcal{T}\circ\mathcal{V}[\![e_2]\!]\ \texttt{else}\ \mathcal{T}\circ\mathcal{V}[\![e_3]\!]\end{aligned}$$

and from lemma 3.4.5 we have $\rho^{cc} \vdash_{\phi^{\mathcal{TV}}} \mathcal{V}[\![e_1]\!] \rightarrow \texttt{true}$ and so

$$\rho^{cc} \vdash_{\phi^{\mathcal{TV}}} \mathcal{T}\circ\mathcal{V}[\![\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3]\!] \rightarrow n_1 + n_2$$

(by rules **Cond.1** and **App.3**) as required.

SCond.2

Result follows in the same way as the previous case, using lemma 3.4.5 and **Cond.2**.

□

3.5 Generalisations of Cost-Functions

In the previous chapter we noted that there were many variations of cost-function definitions possible: cost-functions which count the number of primitive functions, cost functions which give a "micro" result as a sum of symbolic cost-identifiers for every action in the style of [Weg75, Coh82]. There are many other possibilities for deriving functions which compute some property of an operational semantics proof.

The cost-closure technique can be easily adapted to give many of these these variants. The reason for this is that the properties of $\mathcal{V}$ do not depend on the definitions of the cost functions themselves. This enables us to reuse $\mathcal{V}$ to construct other "cost" programs by just supplying alternative versions of $\mathcal{T}$.

As a small example, consider replacing $\mathcal{T}$ by the mapping $\mathcal{TR}[\![\ ]\!]$ defined in figure 3.12.

In this definition, the function <> is an infix list-append. The trace-apply function `tr@` is similar to `c@`, with the zero-cost `0` replaced by the zero-trace, `nil`:

$$(f, cf, n)\ \texttt{tr@}\ e = \begin{cases} cf\ e & \text{if } n = 1\\ \texttt{nil} & \text{otherwise}\end{cases}$$

These definitions together with $\mathcal{V}$ are used to construct trace functions for each f_i:

$$cf_i\ x_1 \ldots x_{n_i} = \texttt{cons}\ \texttt{"}f_i\texttt{"}\ \ \mathcal{TR}\circ\mathcal{V}[\![e_i]\!]$$

The trace functions compute a list of the function names called in the evaluation of the program. In terms of the standard semantics this corresponds to a list of

$$\begin{aligned}
\mathcal{TR}[\![exp' \;\texttt{@}\; e']\!] &= \mathcal{TR}[\![exp']\!] <> \mathcal{TR}[\![e']\!] <> \texttt{(}exp' \;\texttt{tr@}\; e'\texttt{)} \\
\mathcal{TR}[\![\texttt{if}\; e'_1 \;\texttt{then}\; e'_2 \;\texttt{else}\; e'_3]\!] &= \mathcal{TR}[\![e'_1]\!] <> \texttt{if}\; e'_1 \;\texttt{then}\; \mathcal{TR}[\![e'_2]\!] \;\texttt{else}\; \mathcal{TR}[\![e'_3]\!] \\
\mathcal{TR}[\![\texttt{(}exp'\texttt{)}]\!] &= \texttt{(}\mathcal{TR}[\![exp']\!]\texttt{)} \\
\mathcal{TR}[\![\texttt{(}p_i \;\texttt{,}\; cp_i \;\texttt{,}\; m_i\texttt{)}]\!] &= \texttt{nil} \\
\mathcal{TR}[\![\texttt{(}f'_i \;\texttt{,}\; cf_i \;\texttt{,}\; n_i\texttt{)}]\!] &= \texttt{nil} \\
\mathcal{TR}[\![c]\!] &= \texttt{nil}
\end{aligned}$$

Figure 3.12: Trace-Expression Construction Map, $\mathcal{TR}[\![\;]\!]$

the function-name look-ups in the function environment, ordered by a left to right traversal of the proof tree, corresponding to a left to right evaluation order. To do this formally, we would simply need a new version of theorem 3.4.6.

Trace semantics, and related ideas could form the basis of debugging tools, or could be approximated statically to perform analyses along the lines of [Ses89, BH88], and like cost-functions, are amenable to functional programming technology.

3.6 Factoring Higher-Order Cost-Functions

3.6.1 Reasoning about Cost-Functions

An important property of cost-programs is that whilst the cost of evaluating any expression is described in terms of the functions and cost-functions of the cost-program, the *definitions* of the cost-functions are independent of the ways in which they are used by the program. This fact is clearly seen in some of our previous examples where we reason about the performance of individual functions rather than whole programs.

Traditionally, the aim of an analysis is to find a closed-form solution to the cost-equations in terms of some size-measure of the input. In general an exact solution is not possible, and so *upper-bound*, and *average-case* solutions are sought, by making suitable probabilistic assumptions [Fla85, HC88]. These "traditional" methods are largely applicable to reasoning about first-order functions, but break down in the case of functions with function-valued inputs: to reason about such functions "out of context" we would require not only assumptions about the possible values yielded by application of functional arguments, but also about the possible costs incurred by these applications. For our formulation of cost-functions, this might mean giving some probabilistic description of the possible *cost-closures* bound to some variable, which is clearly not a sensible activity!

One solution to this is to only reason about such functions (like `map`) in a particular *context*, *i.e.* when the functional arguments (and hence the cost-closures) are known. Another simple solution is to make sweeping approximations about higher-order parameters. These approximations are of two types:

(i) **Intensional assumptions** involve making assumptions about the cost of applying unknown functional arguments. For example, an expression of the form x_i `c@` *exp* can be replaced by a symbolic constant, to represent an average or upper-bound cost. Replacing such applications by an actual constant can be viewed as a specialisation of cost-functions. For example, the `map` function of section 3.3.3 has the following cost-function:

```
cmap' f x  =  1 + if (null x) then 0
                  else (f c@ (hd x)) + (cmap' f (tl x))
```

The presence of the unknown cost-closure `f` prohibits further manipulation of this expression. If we replace `f c@ (hd x)` by the symbolic constant f_c then we can easily show that

$$\texttt{cmap' f x} \quad \sim \quad \texttt{1 + (}f_c\texttt{ + 1)}n$$

where $n =$ `length x`. The meaning of $\sim$ is then dependent on our interpretation of f_c as a lower, average or upper bound on the cost of applications of `f` to elements of `x`.

(ii) **Extensional assumptions** involve assumptions or approximations about the function-component of an unknown cost-closure, *i.e.* typically an expression of the form x_i `@` *exp*. Particularly useful are *structural* assumptions which arise, for example, in the following way: suppose we have analysed some cost-function `cf x =` e (corresponding to some definition of a function `f`) and we have shown it to be equivalent to some function of it's input length, *i.e.*

```
cf x = g (length x)
```

for some function `g`. Analysing an instance of the form `cf(y @` *exp*`)` where `y` is an unknown, we can make some progress by considering a class of functions (*i.e.* cost-closures) `y` satisfying

$$\texttt{length (y @ }exp\texttt{)} \leq \texttt{length }exp$$

For example `f` may be some sorting function, and we consider the class of contexts in which (the function-component of) `y` is a *filter* (*i.e.* only removes some elements from the list).

3.6.2 Context-Free Cost

When manipulating programs one quite reasonably expects to transform higher-order functions to give more efficient versions without considering specific contexts. The basis of such activities are intuitions about invariant costs associated with such functions: we refer to this component of inherent cost in a higher-order function, which is however independent of the specific functional parameters, as the *context-free* cost. In the remainder of this section we show how it is possible to reason about context-free cost by illustrating how higher-order cost-functions can be factored into a sum of context-free and context-sensitive cost-functions.

Analysing a higher-order function

Consider the following definition of a function `pam` which maps a function over a list, producing the result in reverse order:

```
pam f x  =  if (null x)
            then nil
            else append (pam f (tl x)) (cons (f (hd x)) nil)

append x y = if (null x)
             then y
             else cons (hd x) (append (tl x) y)
```

Deriving the cost expression for `pam` we obtain:

```
cpam f x = 1 + if (null x) then 0
               else cappend (pam f (tl x)) (cons (f @ (hd x)) nil)
                  + cpam f (tl x)
                  + f c@ (hd x)
```

The cost-function for `append` is, as given in chapter 2, equal to

```
cappend x y = 1 + length x
```

Substituting this in the equation for `cpam` gives

```
cpam f x = 1 + if (null x) then 0
               else 1 + length (pam f (tl x))
                      + cpam f (tl x)
                      + f c@ (hd x)
```

The following result is easily shown by induction in `a`:

PROPOSITION **3.6.1** `length (append a b) = length a + length b`

This is used to show that `length (pam f (tl x))` is independent of the functional parameter, *viz.*,

PROPOSITION **3.6.2** `length (pam f a) = length a`

PROOF Induction in `a`:

- Base (`a = nil`): `length (pam f nil) = length nil` { `pam` defn. }
- Induction (`a = cons x xs`):

```
length (pam f (cons x xs))
  =  length (append (pam f xs) (cons (f @ x) nil))
{null, if, hd, tl laws}
  =  length (pam f xs) + length (cons (f @ (hd x)) nil)
{Proposition3.6.1}
  =  length xs + length (cons (f @ (hd x)) nil)
{hypothesis}
  =  length xs + 1
  =  length (cons x xs)
```

□

Substituting this result back into `cpam` gives:

```
cpam f x  = 1 + if (null x) then 0
                else  1 + length (tl x)
                        + cpam f (tl x)
                        + f c@ (hd x)
```

We cannot achieve a closed form expression for this complexity, since the total cost is dependent on the specific function `f`. However, there is useful "context–free" information within this definition, namely the cost information which is independent of the function `f`.

Factoring the cost-function

If we can represent the recursive cost-function `cpam` as a sum of two (recursive) functions, `cpamF` and `cpamS`, where `cpamF` computes the context-**F**ree cost and `cpamS` the context-**S**ensitive cost, then we can potentially derive some closed-form result about the context-free component.

Examining the above equation for `cpam`, we see that the main obstacle is the context-sensitive cost, `f c@ (hd x)`. We begin with the eureka step of defining the function `cpamS` which counts only these components of cost:

```
cpamS f x = if (null x) then 0
            else f c@ (hd x)
               + cpamS f (tl x)
```

Now to complete a factorisation we need to derive a function `cpamF` satisfying

$$\texttt{cpamF f x} = \texttt{(cpam f x) - (cpamS f x)} \qquad (3.59)$$

Starting with this as our definition, we can easily synthesise a recursive version. Instantiating `x` in 3.59 with `nil` and unfolding gives

```
cpamF f nil  = 1 - 0
             = 1
```

Similarly instantiating `x` in 3.59 with `(cons x xs)`, and unfolding gives:

```
cpamF f (cons x xs)  = 1 + length xs
                         + 1
                         + cpam f xs
                         + f c@ x
                         - (f c@ x + cpamS f xs)


                     = 2 + length xs
                         + cpam f xs
                         - cpamS f xs
```

Now we can fold an instance of 3.59 on the right-hand-side to give:

```
cpamF f (cons x xs)  = 2 + length xs
                         + cpamF f xs
```

We now have a recursive definition for `cpamF` in which the parameter `f` is redundant, and can be analysed further by standard recurrence techniques. In this case it is not difficult to show that `cpamF f x` $= \mathcal{O}($`length(x)`$^2)$.

To see an example of how this information may be useful, consider a transformed version of the `pam` function. Program transformation techniques can be used to produce an accumulating parameter solution from the original function (see *e.g.* [Bir84]):

```
pam2 f x = pam' f x nil

pam' f x a = if (null x) then a
             else pam' f (tl x) (cons (f @ (hd x)) a)
```

Analysing these functions, we derive:

```
cpam2 f x = 1 + cpam' f x nil

cpam' f x a = 1 + if (null x) then 0
                  else  cpam' f (tl x) (cons (f @ (hd x)) a)
                        + f c@ (hd x)
```

By redundancy of the third parameter, `cpam' f x a = cpam'' f x` where

```
cpam'' f x  = 1 + if (null x) then 0
                 else cpam' f (tl x)
                          + f c@ (hd x)
```

Again using the factorisation idea, we can use similar factorisations to show that:

```
cpam2 f x    = 1 + cpam'' f x

cpam'' f x  = (cpamF'' f x) + (cpamS'' f x)

cpamS'' f x  = if (null x) then 0
                  else f c@ (hd x) + cpamS'' f (tl x)

cpamF'' f x = 1 + if (null x) then 0
                  else cpamF'' f (tl x)
```

The context-sensitive cost remains unchanged in this accumulating parameter version, however the context-free cost can be shown to be exactly `length x` (versus

$\mathcal{O}($`length(x)`$^2)$ for the original version), thus formally demonstrating the asymptotic improvement of the accumulating parameter version over the original *in any context*.

3.6.3 Generalising Factorisation

Before presenting some more examples, we must consider how the factorisation ideas illustrated in the above example generalise to the analysis of arbitrary higher-order functions.

The degree of success of a factorisation can be measured in terms of obtaining a context-free cost-function which contains no higher-order expressions—in the above expression this is illustrated most clearly by the fact that the unknown functional argument is redundant in the definition of `CpamF`. Also, if we use the symbolic constant f_c to represent an upper bound to the cost-application `f c@ (hd x)` then we can easily show that `cpamS f x` $= f_c.n$ where $n =$ `length x`, and so the factorisation is *complete*, in the sense that `cpamS` could not be any smaller.

In general such complete factorisation will not be possible. In particular it will not be possible when the equations depend on expressions of the form x@e. Note that in the above example we were able to show that `length (pam f (tl x))` was independent of `f`. In an extreme case the outcome of a conditional expression may depend on the value of an unknown function, and so without making some approximating assumptions full factorisation cannot be achieved.

The factorisation in the above example begins with the eureka-step of defining one half of the factorisation pair, and proceeds by synthesising the other half to ensure correctness (in the sense that the sum is equal to the original function). A common goal in the study of program synthesis by transformation is the removal of these so-called ([BD77]) *eureka* steps—see for example [Chi90]. In this case, one possibility for mechanising factorisation is to use type-structure to identify both higher-order parameters and also context-sensitive subexpressions. Since our methods are not restricted to typable functions (see the first example below) we sketch an informal strategy for performing factorisation.

An obvious problem here is that even when the factorisation is achieved, the eureka-step may not define an appropriate half of the factorisation pair (*e.g.* the context-free functions may contain some context-sensitive costs). A strategy for the factorisation-transformation is to take an iterative approach: begin as in the above example by manipulating the functions until the need for factorisation is identified (*i.e.* we encounter expressions of the form x@e where x is not known). A context-sensitive function, `cS`, can then be defined which just computes the (identified) context-sensitive components. After further manipulation, if context-sensitive expressions are still identifiable in the context-free cost-function `cF`, these can be isolated by a second factorisation, producing `cF-` and `cS'`. The context-sensitive functions `cS` and `cS'` can be remerged as a straightforward loop-merging/tupling [Fea82] transformation.

It is straightforward to generalise factorisation to mutually recursive functions: in this case we aim to synthesise two groups of functions each with the same dependency structure. For example, if we have mutually recursive cost-functions `ch` and `cg`, then factorisation derives mutually-recursive context-sensitive functions `chS` and `cgS`, and similarly for the context-free functions.

In the remainder of this section we give some illustrative examples.

3.6.4 Further Examples

The `truth` function

The following function, `truth`, is a function that takes a predicate on booleans, and a number representing the number of arguments that the predicate takes, and tests to see if that predicate always returns `true`.

```
truth p n = if (n = 0) then p
            else (truth (p true) (n-1)) and (truth (p false) (n-1))
```

where `and` is (strict) primitive conjunction. Note that this function is not typable and can go "wrong" if the second argument does not correspond to the number of arguments that the predicate takes. The cost-function derived for this equation is

```
ctruth p n = 1 + if (n = 0) then 0
                 else p c@ true + p c@ false +
                      (ctruth (p @ true) (n-1)) +
                      (ctruth (p @ false) (n-1))
```

Factorisation is straightforward since the context-sensitive components of cost are easily identified: beginning with `ctruthS` defined

```
ctruthS p n = if (n = 0) then 0
              else p c@ true + p c@ false +
                   (ctruthS (p @ true) (n-1)) +
                   (ctruthS (p @ false) (n-1))
```

we can easily synthesise

```
ctruthF p n = 1 + if (n = 0) then 0
                  else (ctruthF (p @ true) (n-1)) +
                       (ctruthF (p @ false) (n-1))
```

Now we can easily show that `ctruthF p n = ctf n`, where

```
ctf n = 1 + if (n = 0) then 0
            else 2 * ctf (n-1)
```

which can be shown to have the solution

$$\texttt{ctf n} = 2^{n+1} - 1$$

The `sumtips` function

The equations below define a function `sumtips` which, given a list of integers, returns a copy whose elements are the sum of the list argument, so for example `sumtips [1,1,2]` = `[4,4,4]`

```
sumtips xs  =  st xs 0 f1
st xs a f   =  if (null xs)
               then (f a)
               else st (tl xs) ((hd xs) + a) (f2 f)
f1 a        =  nil
f2 f a      =  cons a (f a)
```

The derived cost functions are

```
csumtips xs  =  1 + cst xs 0 f1^cc
cst xs a f   =  1 + if (null xs)
                    then (f c@ a)
                    else cst (tl xs) ((hd xs) + a) (f2^cc @ f)
cf1 a        =  1
cf2 f a      =  1 + f c@ a
```

Although the function `cst` has a known context—its call in `sumtips`—it can be analysed independently. As a first step, we can factor the context-free cost "1" and the context-sensitive cost "`(f c@ a)`" to obtain (omitting the derivation details)

```
cst xs a f   =  length xs + cstS xs a f
cstS xs a f  =  if (null xs)
                then (f c@ a)
                else cstS (tl xs) ((hd xs) + a) (f2^cc @ f)
```

In this case there is more context-free cost in `cstS`, but it is not so obviously extracted. There is a further "`length xs`" of context-free cost in `cstS`, as given by:
Proposition **3.6.3**

```
cstS xs a f = length xs + f c@ (a + sum xs)
```

where `sum xs = if (null xs) then 0 else sum (tl xs)`

PROOF *Recursion Induction* ([McC67]): we show that that the proposition satisfies the equation defining `cstS`. Thus we need to show

```
length xs + f c@ (a + sum xs) = if (null xs) then (f c@ a)        (3.60)
                                else length (tl xs)
                                     + (f2^cc @ f) c@ ((hd xs)
                                     + a + sum (tl xs))
```

When `xs` = `nil` then 3.60 holds since `length nil` = 0 and `sum nil` = 0. Otherwise the right-hand side is

```
     if (null xs) then (f c@ a)
     else length (tl xs) + (f2^cc @ f) c@ ((hd xs) + a + sum (tl xs))
  =  length (tl xs) + (f2^cc @ f) c@ ((hd xs) + a + sum (tl xs))
  =  length (tl xs) + cf2 f ((hd xs) + a + sum (tl xs))
  =  length (tl xs) + 1 + f c@ ((hd xs) + a + sum (tl xs))
  =  length xs + f c@ (a + sum xs)
```

□

Returning to the instance of `cstS` in `csumtips`

```
csumtips xs  =  1 + cst xs 0 f1^cc
             =  1 + length xs + cstS xs 0 f1^cc
             =  1 + length xs + length xs + f1^cc c@ (0 + sum xs)
             =  1 + length xs + length xs + 1
             =  2 * (1 + length xs)
```

3.7 Analysis of Call-by-Name via Translations

In this section we discuss a route to the analysis of a call-by-name language, which we call *translation-based analysis*, based on the techniques we have developed thus far. Other more direct means for analysing non-strict evaluation are considered in subsequent chapters; the reader may wish to skip this section on first reading.

It is possible to analyse a call-by-name program using a simulation of the program by a call-by-value one: if we can define a translation from terms in a call-by-name language to strongly equivalent terms in a call-by-value language, then we can analyse the latter program to deduce properties of the former. This idea is seen in [LeM88a] where Le Métayer uses this technique to perform complexity analysis and a novel form of strictness analysis.

In the context of time-analysis, the purpose of the translation is as follows: if we can reduce the problem of time-analysis of a call-by-name program to one of analysing a call-by-value one, then we can apply the cost-closure techniques. However, the obvious problem here is that since we are interested in an *intensional* property, any extensionally correct translation must also preserve the property of interest, and the analysis must be able to distinguish between properties (costs) of the original program, and costs introduced by the translation itself.

Two classes of methods for performing the simulation are identified, both of which come from well-known implementation techniques for functional languages. Finally we discuss the practicalities and limitations of this approach.

3.7.1 Example: A Simple Translation Method

In order to illustrate the method of "translation-based time-analysis" we (informally) describe a simple simulation method. A translation from a language with call-by-name semantics to a language using a call-by-value evaluation mechanism is outlined. We illustrate how the translation together with the call-by-value analysis techniques can be used for analysing call-by-name programs.

A well-known technique for delaying the evaluation of an expression in a call-by-value language is the use of dummy parameters. As mentioned in earlier, there is no evaluation inside the body of a partially-applied function. This means that if we make an expression e into a partially-applied function "$\lambda\texttt{()}.e$" then any evaluation of the expression e will be suspended until we apply a dummy argument corresponding to formal parameter `()`. This technique is well known from the SECD machine [Lan64], where it is used in order to treat the conditional expression as a strict function: the expression

$$\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3$$

is treated as

$$(\texttt{if } e_1 \texttt{ then } \lambda\texttt{()}.e_2 \texttt{ else } \lambda\texttt{()}.e_3)\,\texttt{()}$$

so that the conditional can be implemented in the same manner as any other strict (primitive) function.

A Translation Map

We use this "protecting by lambdas" technique to give a translation map which takes expressions in a call-by-name language to expressions in a call-by-value language.

For simplicity, consider a first-order call-by-name language with syntax as in the previous chapter. We informally assume that, whilst the primitive functions are call-by-value, the operational-semantics is call-by-name.

In the translation, for each function-definition

$$f(x_1, \ldots, x_n) = e$$

we define a simulating function

$$sf\, x_1 \cdots x_n = \mathcal{S}[\![e]\!]$$

where $\mathcal{S}[\![\cdot]\!]$, defined in figure 3.13, takes first-order[3] call-by-name expression to higher-order call-by-value expressions.

For convenience we have also added (as an intermediate step only) a λ-term of the form $\lambda().e$. These are defined for convenience only, and can be "λ-lifted" ([Pey87]) from the definions as follows: expression $\lambda().e$ can be replaced by the function-call

$$f_{new}\ x_1 \cdots x_k$$

where $x_1, \ldots, x_k$ are the free variables of expression e, and we define the new function

$$f_{new}\ x_1 \cdots x_k\ () = e$$

Example

As an example of the simulation, consider the following simple definition:

```
K(x,y) = x
```

[3]It is straightforward to give a higher-order version, but a first-order translation is given here because it simplifies the following illustrative examples of the principal of translation-based time-analysis.

$$
\begin{array}{rcl}
\mathcal{S}[\![f(e_1,\ldots,e_n)]\!] & = & sf\ \lambda().\mathcal{S}[\![e_1]\!]\cdots\lambda().\mathcal{S}[\![e_n]\!] \\
\mathcal{S}[\![p(e_1,\ldots,e_n)]\!] & = & p\ \mathcal{S}[\![e_1]\!]\cdots\mathcal{S}[\![e_n]\!] \\
\mathcal{S}[\![\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3]\!] & = & \texttt{if}\ \mathcal{S}[\![e_1]\!]\ \texttt{then}\ \mathcal{S}[\![e_2]\!]\ \texttt{else}\ \mathcal{S}[\![e_3]\!] \\
\mathcal{S}[\![x]\!] & = & (x\ ()) \\
\mathcal{S}[\![c]\!] & = & c
\end{array}
$$

Figure 3.13: Simulation Map

Under the call-by-name semantics, for any closed expression *exp* we immediately have (rule **Fun**)

$$\texttt{K(7,}\mathit{exp}\texttt{)} \to \texttt{7}$$

Simulating `K` in the call-by-value language we define

$$\texttt{sK x y = } \mathcal{S}[\![\texttt{x}]\!] \texttt{ = (x ())}$$

and

$$\mathcal{S}[\![\texttt{K(7,}\mathit{exp}\texttt{)}]\!] \texttt{ = sK } \lambda().7\ \lambda().\mathcal{S}[\![\mathit{exp}]\!]$$

Removing the λ-expressions from this term we have

$$
\begin{array}{ll}
\texttt{sK A1 A2 where A1 () = 7} \\
\qquad\qquad\qquad\quad\ \texttt{A2 () = } \mathcal{S}[\![\mathit{exp}]\!]
\end{array}
$$

Then (derivation omitted) $\vdash$ `sK A1 A2` $\to$ `7` as required.

To prove the correctness of this simulation we would need to formally give a call-by-name semantics (see, for example, the next chapter). Proofs relating to the simulation are not given here since our main purpose is to illustrate the general principals of a translation-based analysis.

3.7.2 A Translation-Based Time-Analysis

The purpose of translating from a call-by-name language to a call-by-value one is that we can apply the techniques developed so far to analyse the time-cost of the translated program. A potential problem with an arbitrary translation is that the complexity of the translated program may not have any obvious relation to the complexity of the original one. For example, in a call-by-name language a strict function can be safely simulated by a call-by-value version, but since the call-by-value version avoids potential recomputation of the argument, giving a possibly lower complexity for the translated program than the original, this kind of translation does not preserve the property of interest.

In fact, the translations (like $\mathcal{S}$) that are suitable candidates for use in a time-analysis establish a much more straightforward (intensional) relationship between the source (call-by-name) and translated (call-by-value) programs. The basic property of such suitable simulations is that the evaluation of the simulating expressions entails performing *additional* evaluation-steps which are the "administrative" steps of the simulation process, interleaved with steps corresponding to those in the original call-by-name program. Later we will describe two classes of "suitable" translations, which owe there suitability to the fact that they are closely related to standard implementation techniques for non-strict languages.

In the programs obtained by $\mathcal{S}$, the *administrative* steps correspond to the application of the `()`-parameters to resume the evaluation of the λ-guarded expressions. We can easily account for these in our use of the analysis methods of section 3.3 since these steps correspond to the application of the *auxiliary* functions (introduced by λ-lifing), and hence can be discounted.

Let $\hat{\mathcal{S}}[\![e]\!]$ denote the expression $\mathcal{S}[\![e]\!]$ after λ-lifing (and hence after the introduction of some new function-definitions). Now the cost of evaluating some closed expression e in the program-scheme in figure 3.14 is given by the scheme in figure 3.15, where the functions

$$\begin{array}{rcl} A_1\, y_1 \cdots y_{m_1}\, () & = & e'_1 \\ & \cdots & \\ A_p\, y_1 \cdots y_{m_p}\, () & = & e'_p \end{array}$$

are new functions introduced by the λ-lifing process.

Note that in the corresponding cost-function definitions the applications of these functions are not counted as a component of the total cost, *viz.* their right-hand sides do not contain the "`1 +` " subexpression.

3.7.3 Example

We illustrate the method with an example. Firstly we introduce some simplifying notation:

DEFINITION **3.7.1** *For all constants c, let $\overline{c}$ denote the function*

$$\overline{c}\ x = c$$

□

$$
\boxed{
\begin{array}{c}
\begin{array}{rcl}
f_1(x_1, \ldots, x_{n_1}) & = & e_1 \\
 & \vdots & \\
f_k(x_1, \ldots, x_{n_k}) & = & e_k
\end{array} \\
\hline
e
\end{array}
}
$$

Figure 3.14: Program Scheme

$$
\boxed{
\begin{array}{c}
\begin{array}{lcl}
sf_1{}'\ x_1 \ldots x_{n_1} & = & \mathcal{V} \circ \hat{\mathcal{S}}[\![e_1]\!] \\
 & \vdots & \\
sf_k{}'\ x_1 \ldots x_{n_k} & = & \mathcal{V} \circ \hat{\mathcal{S}}[\![e_k]\!] \\
A'_1\ y_1 \cdots y_{m_1}() & = & \mathcal{V} \circ \hat{\mathcal{S}}[\![e'_1]\!] \\
 & \vdots & \\
A'_p\ y_1 \cdots y_{m_p}() & = & \mathcal{V} \circ \hat{\mathcal{S}}[\![e'_p]\!] \\
csf_1\ x_1 \ldots x_{n_1} & = & 1 + \mathcal{T} \circ \mathcal{V} \circ \hat{\mathcal{S}}[\![e_1]\!] \\
 & \vdots & \\
csf_k\ x_1 \ldots x_{n_k} & = & 1 + \mathcal{T} \circ \mathcal{V} \circ \hat{\mathcal{S}}[\![e_k]\!] \\
cA_1\ y_1 \cdots y_{m_1}() & = & \mathcal{T} \circ \mathcal{V} \circ \hat{\mathcal{S}}[\![e'_1]\!] \\
 & \vdots & \\
cA_k\ y_1 \cdots y_{m_p}() & = & \mathcal{T} \circ \mathcal{V} \circ \hat{\mathcal{S}}[\![e'_p]\!]
\end{array} \\
\hline
\mathcal{T} \circ \mathcal{V} \circ \hat{\mathcal{S}}[\![e]\!]
\end{array}
}
$$

Figure 3.15: Simulating Cost-Program Scheme

So, in particular, the function $\lambda().c$ in the cost-program translates to the cost-closure $(\overline{c}\ ,\ \overline{0}\ ,\ 1\ )$, which we will write as $\overline{c}^{cc}$ (see definition 3.3.3).

Now consider the (call-by-name) evaluation of the expression `f(`n`,`m`)` for some values (constants) m and n, where `f` is defined by

```
f(x,y)  =  if x = 0 then 0
           else f(x-1,f(x,y))
```

In figure 3.16 we give the cost-program (including redundant functions) as derived according to figure 3.15. First we note that the second parameter in the definition

```
sf' x y     =  if (x @ ()) = 0 then 0
               else sf' (A1^cc @ x) (A2^cc @ x @ y)
A1' x ()    =  (x @ ()) - 1
A2' x y ()  =  sf' @ x @ y

csf x y     =  1 + (x c@ ())
                 + if (x @ ()) = 0 then 0
                   else csf (A1^cc @ x) (A2^cc @ x @ y)
cA1 x ()    =  (x c@ ())
cA2 x y ()  =  csf x y
```

`csf` $\overline{n}^{cc}$ $\overline{m}^{cc}$

Figure 3.16: Simulating Cost-Program

of `csf` is redundant (since it only appears in the recursive call), so we have

```
csf x y  =  csf' x
csf' x   =  1 + (x c@ ())
              + if (x @ ()) = 0 then 0
                else csf' (A1^cc @ x)
```

PROPOSITION **3.7.2**

$$\texttt{csf'}\ \overline{n}^{cc} = n+1$$

PROOF Mathematical induction on n[4]:

- Base ($n = 0$):

$$\begin{array}{lll}\texttt{csf'}\ \overline{0}^{cc} & = & \texttt{1 + (}\overline{0}^{cc}\texttt{ c@ ())} \\ & & \quad \texttt{+ if (}\overline{0}^{cc}\texttt{ @ ()) = 0 then 0} \\ & & \qquad \texttt{else csf' (A1}^{cc}\texttt{ @ }\overline{0}^{cc}\texttt{)} \\ & = & \texttt{1 + 0} \\ & & \quad \texttt{+ if 0 = 0 then 0} \\ & & \qquad \texttt{else csf' (A1}^{cc}\texttt{ @ }\overline{0}^{cc}\texttt{)} \\ & = & \texttt{1}\end{array}$$

- Inductive case (Hypothesis: $\texttt{csf'}\ \overline{k}^{cc} = k+1$):

$$\begin{array}{lll}\texttt{csf'}\ \overline{k+1}^{cc} & = & \texttt{1 + (}\overline{k+1}^{cc}\texttt{ c@ ())} \\ & & \quad \texttt{+ if (}\overline{k+1}^{cc}\texttt{ @ ()) = 0 then 0} \\ & & \qquad \texttt{else csf' (A1}^{cc}\texttt{ @ }\overline{k+1}^{cc}\texttt{)} \\ & = & \texttt{1 + 0 + csf' (A1}^{cc}\texttt{ @ }\overline{k+1}^{cc}\texttt{)}\end{array}$$

Now for any y we have

$$\begin{array}{lll}\texttt{A1}^{cc}\texttt{ @ }\overline{k+1}^{cc}\texttt{ @ }y & = & \texttt{(}\overline{k+1}^{cc}\texttt{ @ ()) - 1} \\ & = & (k+1)\texttt{ - 1} \\ & = & k \\ & = & \overline{k}^{cc}\texttt{ @ }y\end{array}$$

$$\begin{array}{lll}\texttt{A1}^{cc}\texttt{ @ }\overline{k+1}^{cc}\texttt{ c@ }y & = & \texttt{((}\overline{k+1}^{cc}\texttt{ c@ ())} \\ & = & \texttt{0} \\ & = & \overline{k}^{cc}\texttt{ c@ }y\end{array}$$

and so we have the cost-closure equivalence[5]

$$\texttt{(A1}^{cc}\texttt{ @ }\overline{k+1}^{cc}\texttt{)} \equiv \overline{k}^{cc}$$

[4]The proposition can also be proved via the synthesis of a function `csf''` satisfying

$$\texttt{csf' x} = \texttt{csf'' x (x @ ()) (x c@ ())}$$

but this derivation is somewhat lengthy.

[5]The only way in which cost-closures are used is via `@` or `c@`. If the `fun` and `cost` components are equivalent, then in they are equivalent in all cost-program contexts.

Using this, together with the inductive hypothesis, gives us

$$\begin{aligned}
\texttt{csf'}\ \overline{k+1}^{cc} &= \texttt{1 + csf'}\ (\texttt{A1}^{cc}\ \texttt{@}\ \overline{k+1}^{cc}) \\
&= \texttt{1 + csf'}\ (\overline{k}^{cc}) \\
&= \texttt{1 +}\ (k+1)
\end{aligned}$$

□

3.7.4 Classes of Translation

In this section we broadly classify two methods for translation, whose suitability (in terms of their preservation of cost) is due to their use of standard implementation techniques for non-strict languages.

Closures for Simulation

A *closure* (also known as a *suspension* or *thunk*) is a way of wrapping up an expression together with its environment. This is a standard implementation technique for non-strict functional languages, which allows us to "suspend" the evaluation of an expression.

By creating data structures which model closures, and providing an *eval* function which forces their evaluation, Le Métayer [LeM88a] shows how call-by-value functions can simulate a call-by-name evaluation order. This method is used to perform various analyses, including cost analysis. A problem here is that the analysis counts the steps introduced by the translation itself, which may not be what is required.

The translation-method that we have given, although somewhat simpler than Le Métayer's, falls under the same category of closure-based translation. The difference, and the resulting simplification is due to the fact that we are using the "internal" closure-representation of higher-order-functions in place of explicit closure structures.

Continuations for Simulation

An alternative method for defining a translation is the use of continuations. Continuations were originally introduced [SW71], to give a denotational semantics to languages containing "goto's". They have been found to give a good denotational-based route to implementations (see *e.g.*[Sch86]).

Reynolds, [Rey72], has shown how a continuation-style approach can be used to define an interpreter whose evaluation-order is independent of the evaluation order of the interpreted program.

In Plotkin's "Call-by-Name, Call-by-Value and the λ-calculus" [Plo75], the relation between call-by-name and call-by-value is studied by giving simulations of each language (*i.e.* calling mechanism) by the other, using a method derived from Reynolds' approach. The translation defined in [Plo75], which takes terms in the call-by-name λ-calculus to (simulating) terms in the call-by-value λ-calculus is, from the theoretical viewpoint, a good candidate for a translation-based analysis, exemplified by the elegant relationship established between the reduction of call-by-name terms and the reduction of their call-by-value "simulations"—the reduction steps in the evaluation of the call-by-value terms can be viewed as sequences of: "administrative" reductions (*i.e.* resulting from the simulation) followed by a "genuine" reduction (*i.e.* corresponding to a reduction step of the simulated call-by-name term). The translation is however somewhat impractical for use by hand since it generates complex terms whose time-analysis involves the construction of numerous cost-closures.

3.7.5 Discussion

We have emphasised how time-analysis based on a translation map allows us to use the techniques developed earlier in this chapter, to construct time-equations. The implication of this is that we can reason about complexity without reasoning at the cumbersome operational-semantic level. The operational details are externalised by the translation, thus allowing us to prove or derive properties from the cost functions in the usual "equational" way. This approach therefore seems quite promising, however simple investigation has highlighted some limitations of this approach:

- **Meta-complexity:** the "cogs and wheels" introduced by the translation increases the complexity of the program under analysis to such an extent that the problem of analysis is vastly increased. This is the price paid for externalising the operational details. Furthermore, in order to cope with lazy evaluation (*i.e.* sharing) we would need a much more sophisticated simulation.

- **Compositionality:** an important limitation of the simulation methods outlined are that they are not *compositional*—the cost of evaluating a function cannot be examined independently from the program in which it is used. The cost function corresponding to a (translated) function simply computes the cost of *building a closure*. The translated function must be placed in some context in order to observe its cost-behaviour. A compositional approach to analysis of non-strict evaluation is considered in chapter 5.

3.8 Related Work

In the first part of this chapter we have presented a means of analysing a higher-order strict language, by showing how to construct a functional program which computes the time complexity (in terms of the number of steps executed) of a given program. The derivation is a mechanisable source-to-source translation, and thus can form the basis of a system which can utilise any program transformation or proof technique available to the language in order to reason about the time complexity of higher-order programs, as well as providing a more formal route for functional programmers to reason about (call-by-value) programs. In the remainder of this chapter we discuss related research.

Shultis: On the Complexity of Higher-Order Programs

Analysing the time-complexity of higher-order functions has been considered by Shultis [Shu85]. This study begins with the definition of a *cost-model* via a non-standard denotational semantics. The semantic function has functionality

$$\mathbf{Exp} \to \mathbf{Env} \to \mathbf{Val} \times \mathbf{Cost}$$

where $\mathbf{Val} = \mathbf{Basic} + [\mathbf{Val}^* \to \mathbf{Val} \times \mathbf{Cost}]$ and the $\mathbf{Cost}$ domain is just the positive integers. The denotation pair $\mathbf{Val} \times \mathbf{Cost}$ represents a value, and the associated cost of computing that value.

Because the cost-model is expressed at the level of a denotational metalanguage, an alternative theory is sought in which expressions in the language are manipulated directly. This theory is presented in the form of a logic in which properties about values and costs can be inferred. In the logic, v_e denotes the value of expression e, and t^0_e denotes the *zero-toll* of expression e. The i^{th} toll of an expression is a description of how to obtain the cost of the i^{th} application. So in particular, t^0_e is the cost of evaluating e, and t^1_e is a function which computes the cost of applying e. The axiom for application is illustrative of the logic:

$$\vdash t^0_{(e_1\ e_2)} = t^0_{e_1} + t^0_{e_2} + (t^1_{e_1}\ v_{e_2})$$

Comparison Shultis' cost-model is essentially a denotational counterpart of the step-counting semantics. The cost-component of this semantics is an extension of the standard denotational semantics, but this extension is somewhat arbitrary since it relies on intuitive operational reading of the denotations, whereas the step-count component of our semantics is an externalisation of a precise operational property.

The purpose of the logic is to escape the cumbersome metalanguage, but the logic itself leads to an ambiguous mixture of expressions and meta-expressions. By contrast, our approach (following the style of [LeM85]) of using the language itself as the metalanguage, leads to a succinct formulation of the properties of the cost-model which is provably correct, and inherits the well-studied logic of functional-programs. In [Shu85] the logic is "tested" against the model via an implementation of the semantics—no formal connection is provided between the logic and the model.

There are strong similarities between the axiom for application above, and our scheme for constructing the application cost-expression. A significant difference here is that we do not have any direct counterpart to the higher tolls of an expression, $t^i_{\mathrm{e}}, i > 1$ (which is why we are able to construct a *static* cost-program—see further discussion below). The domain of tolls at the zero-level is just a simple cost; at level $i > 0$, t^i_{e} is a function from

$$\mathrm{Value}_1 \rightarrow \cdots \rightarrow \mathrm{Value}_i \rightarrow \mathrm{Cost}$$

This overall domain of tolls is analogous to the domain of *strictness ladders* from [HY86], where the strictness of a functional argument is described by its simple (level-0) strictness, and its strictness properties at various degrees of application.

Le Métayer: Program Analysis by Program Transformation

In [LeM88a] Le Métayer considers various program analyses (primarily complexity analysis) via an encoding of the problem in the language itself, and thus reducing the analysis to a program transformation problem. The analysis of a higher-order call-by-value functional language is achieved by associating a family of cost-functions with each function in the original program, for which the i^{th} cost-function computes (given i arguments) the cost of applying the function to its i^{th} argument. Thus the cost of some application $(e_1\ e_2 \ldots e_n)$ is the sum

$$C0[e_1] + (C1[e_1]\ e_2) + \cdots + (Cn[e_1]\ e_2 \ldots e_n)$$

where $C_i[e]$ is the i^{th} cost-function of expression e.

Comparison The aims of the above work is to illustrate how a range of (functional) program analysis problems can be encoded in the language under analysis and solved by program transformation techniques. The method we have presented in our higher-order analysis is developed in the style of this approach. We have shown, for example in the factoring-technique, how this approach can be fruitful.

However, Le Métayer's treatment of higher-order functions is much closer to Shultis', where the i^{th} cost-function corresponds to the i-toll of that function. In our approach, the cost-function definitions corresponding to a collection of functions are determined completely statically. In this sense we are much closer to the aims of the above work, since the syntax-directed rules for obtaining the cost functions in [LeM88a] require that cost function definitions are constructed *dynamically* in a way that could not be supported in the language. As an extreme example, consider the following "perverse" identity function

```
id x      =  x
pervid n  =  if (n=0) then id
             else (pervid (n-1) id)
```

If we have an expression of the form (`pervid` m `true`) then we need "m" cost-function definitions for the function `pervid`, since there will be an instance of `pervid` which gets applied to m arguments. Since in general we do not know the value of m, to execute the cost function we must construct the appropriate definitions as we need them. This would require a language with some form of *reflection*, and would consequently be rather more difficult to reason about. Full Hindley-Milner type checking [DM82] would guarantee that within any specific *program* a function is called at finitely many polymorphic instances (the above function is not Hindley-Milner typeable, although it is admissible in some polymorphically-typed functional languages such as HOPE+ [Per88]) however, type-restrictions form only a partial solution to the problem of dynamic cost-function construction.

By contrast, the higher-order analysis presented in this chapter proceeds by constructing (syntactically, statically) cost-functions associated with each function definition in the program. The cost of evaluating any expression is then computed by a term constructed syntactically using the (modified) function-definitions and these cost-functions. So, given the equations above, the reader is invited to confirm that the cost of evaluating *any* expression which uses these definitions requires only one cost-function for each function, and these are not dependent on the use of the functions, merely on their definitions.

In addition, we are able to handle lists of functions (*i.e.* `hd` is potentially higher-order), and untypable functions (such as the `truth`-tester in section 3.6.4), which are not easily handled by the above approach.

Correctness

None of the works cited place any emphasis on proving the correctness of the methods by relating the techniques to a cost-model. We have provided an appropriate cost-model, via an operational semantics, and have proved that the cost-function derivation is correct. As we noted in the previous chapter, the work of Talcott, [Tal85a, Tal85b] is concerned with providing tools for reasoning about intensional properties of programs (like cost). The operational aspects of this framework enable formal reasoning about cost (and other) properties of programs, and the usefulness of expressing these properties as programs (called *derived programs*) is also noted. However, general methods for constructing such programs in the presence of higher-order functions are not achieved.

Non-strict evaluation

In this chapter we have also sketched a technique for constructing time-equations via a translation map, focusing on an example translation from a call-by-name language to a call-by-value one, using the "protecting by lambdas" technique. In theory the translational approach is more generally applicable than just for the analysis of non-strict languages (a claim made in [Ros89]), however in practice such translations are likely to be very complex (and difficult to justify formally) and only useful in the presence of a system which mechanises much of the process of cost-function manipulation.

In the next chapter we develop a much more direct calculus for analysing call-by-name languages than that yielded by the translation method. The key elements of this approach are a more direct calculus based on a simple operational semantics, together with a nonstandard theory of operational approximation.

Chapter 4

An Operational Calculus for Time Analysis

4.1 Introduction

Deriving equations which describe time-cost for a language with non-strict semantics is not straightforward. In the previous chapter we showed how call-by-name evaluation could be analysed in terms of the techniques developed for call-by-value. However, a significant drawback of this approach is the meta-complexity introduced by the translation itself, which makes the subsequent analysis somewhat tedious. In this chapter we will define a simple language with a call-by-name calling mechanism, through which we explore the potential of using the operational definition of time-complexity directly as a calculus for program analysis, rather than going via cost-functions. We construct a "minimalist" semantics for a first-order call-by-name language with lists which leads to a correspondingly simple definition of time, which can be refined to give a calculus with which we can analyse time-cost.

To enable richer forms of equational reasoning within the calculus we require a suitably specialised equivalence-relation between expressions. We develop a non-standard notion of operational approximation, called *cost-simulation*, by analogy with Park's (bi)simulation in CCS. The required property of cost-simulation that makes it central to the calculus is that it is a pre-congruence, and this is proved.

4.2 Syntax

We consider a first-order language with lists, for which we add a case-expression. The justification for introducing the case-expression is that we will be giving the language a call-by-name semantics, and so the typical list-conditional of the form

$$\texttt{if null(x) then } y \texttt{ else } H(\texttt{hd(x)},\texttt{tl(x)})$$

can require `x` to be evaluated three times. Adding a case expression solves this common problem, and gives a closer approximation to call-by-need. The only other non-strict function we need is the list constructor. For this we add an infix syntax, ":". The remaining primitive functions are taken to be strict.

Programs are closed-expressions evaluated in the context of function definitions

$$f_i(x_1, \ldots, x_{n_i}) = e_i$$

Expressions are described by the grammar in figure 4.1.

$$\begin{array}{rcll}
e & ::= & f(e_1, \ldots, e_n) & \text{(function call)} \\
& | & \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 & \text{(conditional)} \\
& | & \texttt{case } e_1 \texttt{ of} & \text{(list-case expression)} \\
& & \qquad\quad \texttt{nil} \;\; \texttt{=>} \;\; e_2 & \\
& & \qquad\quad x : xs \;\; \texttt{=>} \;\; e_3 & \\
& | & e_1 : e_2 & \text{(cons)} \\
& | & (exp) & \text{(parenthesis)} \\
& | & x & \text{(identifier)} \\
& | & c & \text{(constant)}
\end{array}$$

Figure 4.1: Expression Syntax

4.3 Semantics

It is possible to reason about time-complexity of a closed expression by reasoning directly about the "steps" in the evaluation of an expression. This approach requires us to have the machinery of an operational semantics at our fingertips in order to reason in a formal manner. However, the degree of operational reasoning necessary can be minimised by an appropriate choice of semantics. In the operational semantics we have presented so far, we have used an *environment* to bind an identifier to its value. This approach is close to the way we would implement an interpreter for the language. However a more compact description of the semantics can be given using *substitution*, and disposing of the environment altogether: an example of this can be seen in [Plo75], where an "eval"-function which uses substitution rather than explicit closures is defined as an alternative to the evaluation function over SECD-machine states, thus allowing for a more concise description of the properties of the language. We will follow a "substitution" approach in giving a call-by-name operational semantics for the above language.

We will define the semantics via *two* types of evaluation rule: one for evaluation to normal-form, and one for evaluation to head-normal-form [1].

4.3.1 Semantic Rules

For the purpose of the semantics we need to syntactically distinguish between normal and head-normal forms. Normal-forms are the fully evaluated expressions, ranged over by v, v_1, v_2 etc.

$$v ::= c \mid v_1 : v_2$$

Head-Normal forms are simply the constants and any cons-expressions, ranged over by h, h_1, h_2 etc.

$$h ::= c \mid e_1 : e_2$$

In figure 4.2 we define rules which allow us to make judgements of the form: $e \xrightarrow{N} v$ and $e \xrightarrow{H} h$. These can be read as "expression e evaluates to normal-form v" and "expression e evaluates to head-normal-form h" respectively. There is no rule for evaluating a variable—evaluation is only defined over closed-expressions. These rules are presented in figure 4.2, using meta-variable α to range over labels H and N.

Here we give a brief explanation:

- **Function Applications**

  For function application we perform direct substitution of parameters. We use the notation $e\{e'/x\}$ to mean expression e with all occurrences of free variable x replaced by the expression e'. Name-clashes need to be resolved on substitution into case-expressions. A formal definition of substitution is omitted, but is a standard concept from *e.g.* [Bar84].

[1] It is usual to describe the evaluation of functional programs to so-called *weak* head-normal form, see *e.g.* [Pey87], but since we are working with a first-order language weak head-normal forms and head-normal forms are identical.

Fun $$\frac{e_i\{e_1/x_1 \cdots e_{n_i}/x_{n_i}\} \xrightarrow{\alpha} u}{f_i(e_1, \ldots, e_{n_i}) \xrightarrow{\alpha} u}$$

Prim $$\frac{e_1 \xrightarrow{N} c_1 \ \cdots \ e_{n_k} \xrightarrow{N} c_k \quad (v = \mathbf{Apply}(p_k, c_1, \ldots, c_{n_i}))}{p_i(e_1, \ldots, e_{n_k}) \xrightarrow{\alpha} v}$$

Cond $$\frac{e_1 \xrightarrow{N} \texttt{true} \quad e_2 \xrightarrow{\alpha} u}{\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \xrightarrow{\alpha} u} \qquad \frac{e_1 \xrightarrow{N} \texttt{false} \quad e_3 \xrightarrow{\alpha} u}{\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \xrightarrow{\alpha} u}$$

Cons $$\frac{e_1 \xrightarrow{N} v_1 \ , \ e_2 \xrightarrow{N} v_2}{e_1 : e_2 \xrightarrow{N} v_1 : v_2} \qquad \frac{}{e_1 : e_2 \xrightarrow{H} e_1 : e_2}$$

Const $$\frac{}{c \xrightarrow{\alpha} c}$$

Case $$\frac{e_1 \xrightarrow{H} \texttt{nil} \quad e_2 \xrightarrow{\alpha} u}{\left(\begin{array}{lll} \texttt{case } e_1 \texttt{ of} & & \\ \quad\quad \texttt{nil} & \texttt{=>} & e_2 \\ \quad\quad x : xs & \texttt{=>} & e_3 \end{array}\right) \xrightarrow{\alpha} u} \qquad \frac{e_1 \xrightarrow{H} e_h : e_t \ , \ e_3\{e_h/x, e_t/xs\} \xrightarrow{\alpha} u}{\left(\begin{array}{lll} \texttt{case } e_1 \texttt{ of} & & \\ \quad\quad \texttt{nil} & \texttt{=>} & e_2 \\ \quad\quad x : xs & \texttt{=>} & e_3 \end{array}\right) \xrightarrow{\alpha} u}$$

Figure 4.2: Dynamic Semantics

We assume the primitive functions are strict functions on constants, and are defined by some partial function **Apply**.

- **Conditional**

  The conditional is essentially the same as the call-by-value version, and is defined by two rules, selected according to the value of the condition.

- **Case-Expression**

  To evaluate a case-expression, we must evaluate the list-expression e_1 to determine which branch to take. However, regardless of the amount of result we require, we do not wish to evaluate the expression any further than the first cons-node.

Note that in the rules for the conditional, the condition is evaluated to normal-form. In fact, we could evaluate to head-normal form in this case (see Lemma 4.6.8), and in doing so it would be sometimes possible to give better information in the case of an undefined expression by including error-values (*e.g.* consider an expression of the form `if` $e : e'$ `then` e_2 `else` e_3). For our purposes the rules presented will be sufficient.

4.4 Deriving Time-Equations

We wish to reason about the time-cost of evaluating an expression. We again express this property in terms of the number of non-primitive function calls occurring in the evaluation of the expression. For the operational semantics given, this property corresponds to the number of instances of the rule **Fun** in the proof of $e \xrightarrow{N} v$ for some closed expression $e$and value v, whenever such a proof exists.

Let $S, S_1, S_2 \ldots$ range over judgements of the form $e \xrightarrow{H} h$ and $e \xrightarrow{N} v$.

Definition **4.4.1** *Let* $T(\Delta)$ *denote the number of instances of rule* **Fun** *in a given proof* Δ*.*

$\Box$

As before we can define T inductively in the structure of the proof, according to the last rule applied:

$$T\left(\frac{\Delta_1, \ldots, \Delta_k}{S} \quad \mathbf{r}\right) = \begin{cases} 1 + T(\Delta_1) + \cdots + T(\Delta_k) & \text{if } \mathbf{r} = \mathbf{Fun} \\ T(\Delta_1) + \cdots + T(\Delta_k) & \text{otherwise} \end{cases}$$

$$T(S) = 0 \text{ if } S \text{ is an instance of an axiom}$$

4.4.1 A Refined Definition

In the case of the semantics we have given here, we can abstract away from the structure of the proof, and define this property in terms of the structure of *expressions*, since the last rule in the proof of some judgement S is determined by the expression-syntax (hence *structural* operational-semantics). In order to define the time-cost in this manner, we give equations for $\langle e \rangle^N$, the cost of computing the normal-form, and $\langle e \rangle^H$, the cost of computing the head-normal-form of expression e. The equations for $\langle\rangle^N$ and $\langle\rangle^H$ are given in figure 4.3, where α ranges over labels N and H.

$$
\begin{array}{rcl}
\langle f_i(e_1,\ldots,e_{n_i})\rangle^\alpha & = & 1 + \langle e_i\{e_1/x_1,\ldots,e_{n_i}/x_{n_i}\}\rangle^\alpha \\[2ex]
\langle p(e_1,\ldots,e_k)\rangle^\alpha & = & \langle e_1\rangle^N + \cdots + \langle e_k\rangle^N \\[2ex]
\langle \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3\rangle^\alpha & = & \langle e_1\rangle^N + \begin{cases} \langle e_2\rangle^\alpha & \text{if } e_1 \xrightarrow{N} \texttt{true} \\ \langle e_3\rangle^\alpha & \text{if } e_1 \xrightarrow{N} \texttt{false} \end{cases} \\[2ex]
\left\langle \begin{array}{lll} \texttt{case}\ e_1\ \texttt{of} & & \\ \quad\ \texttt{nil} & \texttt{=>} & e_2 \\ \quad\ x : xs & \texttt{=>} & e_3 \end{array} \right\rangle^\alpha & = & \begin{array}{l} \langle e_1\rangle^H \\ + \begin{cases} \langle e_2\rangle^\alpha & \text{if } e_1 \xrightarrow{H} \texttt{nil} \\ \langle e_3\{e_h/x, e_t/xs\}\rangle^\alpha & \text{if } e_1 \xrightarrow{H} e_h : e_t \end{cases} \end{array} \\[2ex]
\langle e_1 : e_2\rangle^N & = & \langle e_1\rangle^N + \langle e_2\rangle^N \\
\langle e_1 : e_2\rangle^H & = & \langle c\rangle^\alpha \ = \ 0
\end{array}
$$

Figure 4.3: Derived Time-Equations

Proposition **4.4.2** *For all expressions e, if Δ is a proof of $e \xrightarrow{\alpha} u$, for some u, then*

$$T(\Delta) = n \Rightarrow \langle e\rangle^\alpha = n$$

The proof of this proposition is a straightforward induction in the structure of Δ.

4.5 Direct Time Analysis

In the time analysis presented in the previous chapters, we focused on the construction of programs to compute the value of $\langle e\rangle^N$. This leads naturally to a compositional approach in which a cost-function is constructed corresponding to each function definition. As we shall see, this compositionality does not come so easily for non-strict languages. First, however, we illustrate that the definitions in figure 4.3 are sufficient to reason directly about the cost of evaluating closed-expressions in this simple language[2].

4.5.1 Example

Consider the functions over lists given in 4.4.

```
append(xs,ys)   =  case  xs of
                         nil => ys
                         h:t => h:append(t,ys)

reverse(xs)     =  case  xs of
                         nil => nil
                         h:t => append(reverse(t), h:nil)
head(xs)        =  case  xs of
                         nil => undefined
                         h:t => h
```

Figure 4.4: Some list-manipulating functions

Now we wish to consider the cost of evaluating the expression

$$\texttt{head(reverse(}v\texttt{))}$$

which computes the last element of some non-empty list-value v. Applying the definitions in figure 4.3

$$\langle \texttt{head(reverse(}v\texttt{))} \rangle^N = 1 + \langle \texttt{reverse(}v\texttt{)} \rangle^H + \langle e_h \rangle^N$$

where $\texttt{reverse(}v\texttt{)} \xrightarrow{H} e_h : e_t$ for some e_h, e_t. It is not hard to show that e_h is a value, and hence $\langle e_h \rangle^N = 0$. Now

$$\langle \texttt{reverse(}v\texttt{)} \rangle^H = 1 + \langle v \rangle^H + \langle \texttt{append(reverse(}e_t\texttt{),}e_h\texttt{:nil)} \rangle^H$$

[2]Acknowledgement: Richard Bird originally pointed out to me that the time-complexity of closed-expressions in a simple call-by-name language could be effectively described and analysed by some relatively simple equations—our main aim here is to show that from a suitably simplified semantic description we can derive a sufficiently concise calculus to perform time analysis

where $v \xrightarrow{H} e_h : e_t$ for some e_h, e_t. Since v is a value of the form $v_1 : v'$, we have

$$\begin{aligned}
&\langle \texttt{reverse(}v\texttt{)}\rangle^H \\
&= 1 + \langle \texttt{append(reverse(}v'\texttt{),}v_1\texttt{:nil)}\rangle^H \\
&= 1 + 1 + \langle \texttt{reverse(}v'\texttt{)}\rangle^H \\
&\quad + \begin{cases} \langle \texttt{nil}\rangle^H & \text{if } \texttt{reverse(}v'\texttt{)} \xrightarrow{H} \texttt{nil} \\ \langle e_h : \texttt{append(}e_2\texttt{,}v\texttt{:nil)}\rangle^H & \text{if } \texttt{reverse(}v'\texttt{)} \xrightarrow{H} e_h : e_t \end{cases} \\
&= 2 + \langle \texttt{reverse(}v'\texttt{)}\rangle^H
\end{aligned}$$

We now have the recurrence equations:

$$\begin{aligned}
\langle \texttt{reverse(nil)}\rangle^H &= 1 \\
\langle \texttt{reverse(}v : vs\texttt{)}\rangle^H &= 2 + \langle \texttt{reverse(}vs\texttt{)}\rangle^H
\end{aligned}$$

whose solution is $\langle \texttt{reverse(}v\texttt{)}\rangle^H = 1 + 2n$, where n is the length of the list v. Thus we have a total cost of

$$\langle \texttt{head(reverse(}v\texttt{))}\rangle^N = 2(1+n)$$

where n is the length of the list v, *i.e.* linear time complexity (cf. quadratic for the call-by-value reading)

4.5.2 Example

Consider the following (somewhat nonstandard) definition of fibonacci:

```
fib(n)    =  f(n,0)
f(n, r)   =  if n=0
             then 1
             else r + f(n-1, f(n-2,0))
```

Consider the time to compute an instance of `fib`:

$$\begin{aligned}
\langle \texttt{fib(}k\texttt{)}\rangle^N &= 1 + \langle \texttt{f(}k\texttt{, 0)}\rangle^N \\
\langle \texttt{f(}k\texttt{, 0)}\rangle^N &= 1 + \left\langle \begin{array}{l}\texttt{if } k\texttt{=0 then 1} \\ \texttt{else 0 + f(}k\texttt{-1, f(}k\texttt{-2,0))}\end{array}\right\rangle^N \\
&= 1 + \begin{cases} \langle \texttt{1}\rangle^N & \text{if } k\texttt{=0} \xrightarrow{N} \texttt{true} \\ \langle \texttt{0 + f(}k\texttt{-1, f(}k\texttt{-2,0))}\rangle^N & \text{if } k\texttt{=0} \xrightarrow{N} \texttt{false} \end{cases} \\
&= 1 + \begin{cases} 0 & \text{if } k\texttt{=0} \xrightarrow{N} \texttt{true} \\ \langle \texttt{f(}k\texttt{-1, f(}k\texttt{-2,0))}\rangle^N & \text{if } k\texttt{=0} \xrightarrow{N} \texttt{false} \end{cases}
\end{aligned}$$

Instantiating k we get

$$\begin{aligned}
\langle \texttt{f(0, 0)} \rangle^N &= 1 && (4.1)\\
\langle \texttt{f(1, 0)} \rangle^N &= 1 + \langle \texttt{f(0, f(1-2,0))} \rangle^N &&\\
&= 2 && (4.2)\\
\langle \texttt{f(}k+2\texttt{, 0)} \rangle^N &= 1 + \langle \texttt{f(}k+1\texttt{, f(}k\texttt{,0))} \rangle^N &&\\
&= 1 + 1 + \langle \texttt{f(}k\texttt{,0)} \rangle^N + \langle \texttt{f(}k\texttt{, f(}k-1\texttt{,0))} \rangle^N && (4.3)
\end{aligned}$$

Now $\langle \texttt{f(}k+1\texttt{, 0)} \rangle^N = 1 + \langle \texttt{f(}k\texttt{, f(}k-1\texttt{,0))} \rangle^N$, so

$$\langle \texttt{f(}k+2\texttt{, 0)} \rangle^N = 1 + \langle \texttt{f(}k\texttt{,0)} \rangle^N + \langle \texttt{f(}k+1\texttt{, 0)} \rangle^N \qquad (4.4)$$

Equations 4.1, 4.2 and 4.4 give a recurrence-relation that can be solved (exactly) using standard techniques (see *e.g.*, [GKP89]); the asymptote is

$$\langle \texttt{f(}k\texttt{, 0)} \rangle^N = \mathcal{O}(k^{(1+\sqrt{5})/2})$$

4.6 A Theory of Cost-Simulation

The above example (`fib`) highlights some potential problems in reasoning about cost using the equations for $\langle\cdot\rangle^H$ and $\langle\cdot\rangle^N$; there are several simplifications which have been used which although in this example are correct, are not generally applicable[3]. If we know that two expressions are equivalent, say e_1 and e_2, it might be tempting to (incorrectly) assume that, $\langle e_1 \rangle^N$ and $\langle e_2 \rangle^N$ will be equivalent. However, it is easy to see that ordinary equational reasoning is not valid—we expect, with any reasonable definition of extensional equivalence that an expression and its normal-form (assuming one exists) will be equivalent, but we certainly would not expect their computational costs to be the same! However, in the above example we have used simple equalities such as (line 4.3) $(k+1)\texttt{-1} = k$ which *are* valid equations for use in the simplification of cost-expressions.

This section is devoted to developing a stronger notion of equivalence, *cost equivalence* which respects cost, and allows a richer form of equational reasoning on expressions within the calculus. The theory of cost-equivalence is interesting in its own right, although the details of its development can be skipped on first reading—however the implication of cost-equivalence with respect to the time-equations (corollary 4.6.15), and the example cost-equivalence laws (proposition 4.6.16) should be noted.

[3]We are not referring here to the solution of recurrence relations, which is a relatively well-studied area.

4.6.1 Motivation

In order to justify the simple instances of equational reasoning in the above example we need to do some further work. Let $C[\,]$ be a *context*, *i.e.* a term with a single hole. Then the "shortcuts" taken in the above example would be justified if we had the following:

PROPOSITION **4.6.1** *If* $p(c_1, \ldots, c_k) \overset{N}{\to} v$ *then*

$$\langle C[p(c_1, \ldots, c_k)] \rangle^N = \langle C[v] \rangle^N$$

As an intuition of why this proposition holds (the proof comes later as an instance of a more general theorem), suppose Φ is the proof of

$$C[p(c_1, \ldots, c_k)] \overset{N}{\to} v'$$

for some value v'. It should be (intuitively) clear that $C[v] \overset{N}{\to} v'$, and that it's proof, Φ', is identical to Φ, except for zero or more instances in Φ of the sub-proof

$$\frac{c_1 \overset{N}{\to} c_1 \ , \ldots , \ c_k \overset{N}{\to} c_k}{p(c_1, \ldots, c_k) \overset{\alpha}{\to} v} \ \textbf{Prim}$$

are replaced by the axiom $v \overset{N}{\to} v$. And since the proofs Φ and Φ' are only different in sub-proofs not containing rule **Fun** then from the definitions of $\langle \cdot \rangle^H$ and $\langle \cdot \rangle^N$ we must have

$$\langle C[p(c_1, \ldots, c_k)] \rangle^N = \langle C[v] \rangle^N$$

We aim to generalise this substitution property further to arbitrary expressions that have non-zero cost; what we would like is the (weakest) equivalence relation $=_c$ which satisfies

$$e =_c e' \Longrightarrow \langle C[e] \rangle^\alpha = \langle C[e'] \rangle^\alpha$$

i.e. we need to find a form of *contextual congruence* relation. To develop this general congruence relation, we use a notion of *simulation* similar to the various simulations developed in Milner's calculus of communicating systems [Mil89]. In the theory of concurrency a central idea is that processes which cannot be distinguished by observation should be identified. This "observational" viewpoint is adopted in Abramsky's treatment of the "lazy" λ-calculus [Abr90b], where an operational equivalence called *applicative bisimulation* is introduced. In the lazy λ-calculus, the observable properties are just the convergence of untyped lambda-terms. For our purposes we need to treat cost as an observable component of the evaluation process, and so we develop a suitable notion of *cost-(bi)simulation*.

4.6.2 Cost-Simulation

As we have seen, the partial functions $\xrightarrow{H}$ and $\xrightarrow{N}$ together with $\langle\cdot\rangle^N$ and $\langle\cdot\rangle^H$ are not sufficient to completely characterise the cost-behaviour of expressions in all contexts. We need to characterise possibly infinite "observations" on expressions which arise in our language because of the non-strict list-constructor (*c.f.* untyped weak head-normal forms in [Abr90b]). The equivalence we develop corresponds intuitively to a notion of *observational equivalence* under all *experiments*. In this context an experiment corresponds to an evaluation using $\xrightarrow{H}$, and the observable properties are the constant or cons-cell produced (but not the expressions that lie "under" the cons), and the cost of its production.

Roughly speaking, the equivalence we develop satisfies:

> *e and e' are equivalent iff $\langle e\rangle^H = \langle e'\rangle^H$ and their head-normal-forms are either identical, or they are cons-expressions whose corresponding components are equivalent.*

Unfortunately, although this is a property that we would like our equivalence to obey, it does not constitute a definition (to see why, note that we do not only wish to relate expressions having normal-forms), so following [Mil83] we use a technique due to Park [Par80] for identifying processes—the notion of a *bisimulation* and its related proof technique. We will develop the equivalence relation we require in terms of preorders called *cost-simulations*—we will then say that two expressions are cost-equivalent if they simulate each other.

We begin with our basic notion of simulation, analagous to Park's (bi)simulation. To simplify our presentation we add some notation:

Definition **4.6.2** *If $\mathcal{R}$ is a binary relation on closed-expressions, then $\mathcal{R}^{\because}$ is the binary relation on head-normal-forms such that*

$$(h\ \mathcal{R}^{\because}\ h')\ \textit{iff}\quad \begin{array}{ll} \textit{either} & h = h' = c \textit{ for some constant } c \\ \textit{or} & h = e_1 : e_2,\ h' = e'_1 : e'_2 \textit{ and} \\ & e_1\ \mathcal{R}\ e'_1,\ \textit{ and } e_2\ \mathcal{R}\ e'_2 \end{array}$$

□

Definition **4.6.3 (Cost-Simulation)** *A binary relation on closed-expressions, $\mathcal{R}$ is a* **cost-simulation** *if, whenever $e\ \mathcal{R}\ e'$*

$$e \xrightarrow{H} h \Longrightarrow (e' \xrightarrow{H} h' \textit{ and } \langle e\rangle^H = \langle e'\rangle^H \textit{ and } h\ \mathcal{R}^{\because}\ h') \tag{4.5}$$

□

DEFINITION **4.6.4** *Let $\mathcal{F}(\mathcal{R})$ be the set of pairs (e, e') satisfying* (4.5)

□

FACT **4.6.5**

- *$\mathcal{F}$ is monotonic, ie, $\mathcal{R} \subseteq \mathcal{S} \Longrightarrow \mathcal{F}(\mathcal{R}) \subseteq \mathcal{F}(\mathcal{S})$*
- *$\mathcal{S}$ is a cost-simulation iff $\mathcal{S} \subseteq \mathcal{F}(\mathcal{S})$*

DEFINITION **4.6.6** *Let $\preceq$ denote the maximum cost-simulation*

$$\bigcup\{\mathcal{S} : \mathcal{S} \subseteq \mathcal{F}(\mathcal{S})\}$$

□

PROPOSITION **4.6.7** *$\preceq$ is the maximal fixed-point of $\mathcal{F}$.*

PROOF Since $\preceq$ is a cost-simulation, $\preceq \subseteq \mathcal{F}(\preceq)$. Monotonicity of $\mathcal{F}$ implies that $\mathcal{F}(\preceq) \subseteq \mathcal{F}(\mathcal{F}(\preceq))$, so $\mathcal{F}(\preceq)$ is also a cost-simulation. But since $\preceq$ is the largest cost-simulation, $\mathcal{F}(\preceq) \subseteq \preceq$. Hence $\mathcal{F}(\preceq) = \preceq$. Also $\preceq$ must be the largest fixed point since any other fixed point is a cost-simulation. □

With these results we have the following useful proof technique: to show that $e \preceq e'$, it is necessary and sufficient to show that *any* cost-simulation contains (e, e'). This technique will be used later in the proof that $\preceq$ is a precongruence.

4.6.3 Relating $\xrightarrow{N}$ and $\xrightarrow{H}$

The above definition of cost-simulation is described in terms of evaluation to head-normal-form only. For this to be sufficient to describe properties of evaluation to normal-form we need some properties relating reductions $\xrightarrow{N}$ and $\xrightarrow{H}$.

LEMMA **4.6.8** *For all closed expressions e, and constants c,*

$$e \xrightarrow{N} c \textit{ iff } e \xrightarrow{H} c \textit{ and } \langle e \rangle^N = \langle e \rangle^H$$

PROOF Straightforward induction in the structure of the proofs of $e \xrightarrow{N} c$ and $e \xrightarrow{H} c$, details omitted. □

Lemma **4.6.9** *For all closed expressions e, $e \xrightarrow{N} v$ iff $e \xrightarrow{H} h$ and $h \xrightarrow{N} v$ and $\langle e \rangle^H + \langle h \rangle^N = \langle e \rangle^N$ for some head-normal-form h.*

proof (⇒) Induction on the structure of the proof of $e \xrightarrow{N} v$. We give an illustrative case:

$$\boxed{\textbf{Fun} \quad \frac{e_i\{e_1/x_1,\ldots,e_{n_i}/x_{n_i}\} \xrightarrow{N} v}{f_i(e_1,\ldots,e_{n_i}) \xrightarrow{N} v}}$$

In this case we are required to prove that, for some h,

$$f_i(e_1,\ldots,e_{n_i}) \xrightarrow{H} h \qquad (4.6)$$

$$h \xrightarrow{N} v \qquad (4.7)$$

$$\langle f_i(e_1,\ldots,e_{n_i}) \rangle^H + \langle h \rangle^N = \langle f_i(e_1,\ldots,e_{n_i}) \rangle^N \qquad (4.8)$$

Since $e_i\{e_1/x_1,\ldots,e_{n_i}/x_{n_i}\} \xrightarrow{N} v$ by a smaller proof, the induction hypothesis gives us that there is an h such that

$$e_i\{e_1/x_1,\ldots,e_{n_i}/x_{n_i}\} \xrightarrow{H} h \qquad (4.9)$$

$$h \xrightarrow{N} v \qquad (4.10)$$

$$\langle e_i\{e_1/x_1,\ldots,e_{n_i}/x_{n_i}\} \rangle^H + \langle h \rangle^N = \langle e_i\{e_1/x_1,\ldots,e_{n_i}/x_{n_i}\} \rangle^N \qquad (4.11)$$

We can conclude 4.6 from 4.9 using rule **Fun**, and 4.7 from 4.10. Finally, for 4.8

$$\begin{aligned} \langle f_i(e_1,\ldots,e_{n_i}) \rangle^N &= 1 + \langle e_i\{e_1/x_1,\ldots,e_{n_i}/x_{n_i}\} \rangle^N && (\text{def. } \langle\rangle^N) \\ &= 1 + \langle e_i\{e_1/x_1,\ldots,e_{n_i}/x_{n_i}\} \rangle^H + \langle h \rangle^N && (\text{from 4.11}) \\ &= \langle f_i(e_1,\ldots,e_{n_i}) \rangle^H + \langle h \rangle^N && (\text{def. } \langle\rangle^H) \end{aligned}$$

The other cases follow in a similar manner. Proof in the ⇐ direction is again a routine inductive proof on the structure of the inference $e \xrightarrow{H} h$. □

Lemma **4.6.10** *If $e \preceq e'$ then*
if $e \xrightarrow{N} u$ then $e' \xrightarrow{N} u$ and $\langle e \rangle^N = \langle e' \rangle^N$

proof Induction on the structure of value u.

- $u \equiv c$
  If $e \xrightarrow{N} c$ then $e \xrightarrow{H} c$ from lemma 4.6.8. Then definition 4.6.3gives $e' \xrightarrow{H} c$ and $\langle e \rangle^H = \langle e' \rangle^H$, and then from lemma 4.6.8

$$\langle e \rangle^N = \langle e \rangle^H = \langle e' \rangle^H = \langle e' \rangle^N$$

  as required.

- $u \equiv v_1 : v_2$
  Suppose $e \overset{N}{\rightarrow} v_1 : v_2$. The induction hypothesis gives: for all e_a, e_b such that $e_a \preceq e_b$

$$\begin{array}{lcl} e_a \overset{N}{\rightarrow} v_1 & \Rightarrow & e_b \overset{N}{\rightarrow} v_1 \;\&\; \langle e_a \rangle^N = \langle e_b \rangle^N \\ e_a \overset{N}{\rightarrow} v_2 & \Rightarrow & e_b \overset{N}{\rightarrow} v_2 \;\&\; \langle e_a \rangle^N = \langle e_b \rangle^N \end{array}$$

From lemma 4.6.9 we know that, for some e_1, e_2

$$\begin{array}{l} e \overset{H}{\rightarrow} e_1 : e_2 \\ e_1 : e_2 \overset{N}{\rightarrow} v_1 : v_2 \\ \langle e \rangle^N = \langle e \rangle^H + \langle e_1 : e_2 \rangle^N \end{array}$$

which implies that

$$\begin{array}{l} e_1 \overset{N}{\rightarrow} v_1 \\ e_2 \overset{N}{\rightarrow} v_2 \\ (\langle e \rangle^N = \langle e \rangle^H + \langle e_1 \rangle^N + \langle e_2 \rangle^N) \end{array} \tag{4.12}$$

Since $e \preceq e'$ we know that $e' \overset{H}{\rightarrow} e'_1 : e'_2$ for some e'_1, e'_2 such that $e_1 \preceq e'_1$ and $e_2 \preceq e'_2$. Now we have, by an instance of the induction hypothesis that, $i = 1, 2$

$$e'_i \overset{N}{\rightarrow} v_i \;\&\; \langle e_i \rangle^N = \langle e'_i \rangle^N \tag{4.13}$$

from which we have, by rule **Cons** that $e'_1 : e'_2 \overset{N}{\rightarrow} v_1 : v_2$, and so $e' \overset{N}{\rightarrow} v_1 : v_2$ from lemma 4.6.9 ($\Leftarrow$).

Now it only remains to show that $\langle e \rangle^N = \langle e' \rangle^N$:

$$\begin{array}{rcll} \langle e \rangle^N & = & \langle e \rangle^H + \langle e_1 \rangle^N + \langle e_2 \rangle^N & \text{(from 4.12)} \\ & = & \langle e \rangle^H + \langle e'_1 \rangle^N + \langle e'_2 \rangle^N & \text{(from 4.13)} \\ & = & \langle e' \rangle^H + \langle e'_1 : e'_2 \rangle^N & \text{(def. } \preceq, \langle\rangle^N) \\ & = & \langle e' \rangle^N & \text{(prop. 4.6.9, } \Leftarrow) \end{array}$$

□

4.6.4 Precongruence

Now we are ready to prove the key property of cost-simulation: cost-simulation is a precongruence, *i.e.* that it is substitutive.

Some notation:

For convenience we abbreviate the indexed family of expressions $\{e_j : j \in J\}$ (for some J) by $\tilde{e}$. Similarly we will abbreviate the substitution $\{e_j/x_j : j \in J\}$ by $\{\tilde{e}/\tilde{x}\}$, and when, for all $j \in J$, $(e_j \mathrel{\Re} e'_j)$ for some relation $\Re$, we write $(\tilde{e} \mathrel{\Re} \tilde{e}')$.

Theorem **4.6.11 (Precongruence)** *If $\tilde{e} \preceq \tilde{e}'$ for some closed expressions $\tilde{e}, \tilde{e}'$, then for all expressions e containing at most variables $\tilde{x}$*

$$e\{\tilde{e}/\tilde{x}\} \preceq e\{\tilde{e}'/\tilde{x}\}$$

proof The relation $\Re$, defined below, is such that $(e\{\tilde{e}/\tilde{x}\}, e\{\tilde{e}'/\tilde{x}\}) \in \Re$ whenever $\tilde{e} \preceq \tilde{e}'$ and e contains at most variables $\tilde{x}$. Lemma 4.6.13 (below) establishes that $\Re$ is a cost-simulation, *i.e.* that $\Re \subseteq \preceq$, and the result follows by Park induction. □

Definition **4.6.12** *$\Re$ is defined to be the relation*

$$\Re = \{(e\{\tilde{e}/\tilde{x}\}, e\{\tilde{e}'/\tilde{x}\}) : e \textit{ contains at most variables } \tilde{x}, \tilde{e} \preceq \tilde{e}'\}$$

□

Lemma **4.6.13** *$\Re$ is a cost-simulation.*

proof Assume that $\tilde{e} \preceq \tilde{e}'$, for some closed expressions $\tilde{e}$, $\tilde{e}'$. Let substitutions $\sigma = \{\tilde{e}/\tilde{x}\}$ and $\sigma' = \{\tilde{e}'/\tilde{x}\}$, and assume that e is some expression containing at most variables $\tilde{x}$. The lemma requires us to prove that

$$e\sigma \xrightarrow{H} h \quad \Rightarrow \quad (\ e\sigma' \xrightarrow{H} h' \quad \& \quad \langle e\sigma\rangle^H = \langle e\sigma'\rangle^H \quad \& \quad h \mathrel{\Re^{\cdot}} h')$$

We prove this by induction on the structure of the inference of $e\sigma \xrightarrow{H} h$. We consider cases according to the structure of expression e.

$\boxed{e \equiv x}$

Since $x \in \tilde{x}$ we have $x\sigma \preceq x\sigma'$ and the result follows from the definition of $\preceq$.

$$\boxed{e \equiv \left(\begin{array}{lll} \texttt{case}\ e_1\ \texttt{of} & & \\ \quad\quad \texttt{nil} & \texttt{=>} & e_2 \\ \quad\quad x : xs & \texttt{=>} & e_3 \end{array}\right)}$$

Assume, *w.l.o.g.* that $\{x, xs\} \cap \tilde{x} = \emptyset$. This case breaks into two sub-cases according to

(i) $e_1\sigma \xrightarrow{H} \texttt{nil}$

(ii) $e_1\sigma \xrightarrow{H} e_h : e_t$

We will only prove the second more difficult case:

The last inference must have the form

$$\frac{e_1\sigma \xrightarrow{H} e_h : e_t \quad e_3\sigma\{e_h/x, e_t/xs\} \xrightarrow{H} h}{\left(\begin{array}{lrl} \texttt{case}\; e_1\sigma\; \texttt{of} & & \\ \qquad\qquad \texttt{nil} & \texttt{=>} & e_2\sigma \\ \qquad\qquad x : xs & \texttt{=>} & e_3\sigma \end{array}\right) \xrightarrow{H} h} \ \textbf{Case.cons}$$

for some h, e_h, e_t. We are required to prove that

$$\left(\begin{array}{lrl} \texttt{case}\; e_1\sigma'\; \texttt{of} & & \\ \qquad\qquad \texttt{nil} & \texttt{=>} & e_2\sigma' \\ \qquad\qquad x : xs & \texttt{=>} & e_3\sigma' \end{array}\right) \xrightarrow{H} h' \qquad (4.14)$$
$$\textit{where}\ (h\ \Re^{\cdot}\ h')$$

$$\left\langle \left(\begin{array}{lrl} \texttt{case}\; e_1\sigma\; \texttt{of} & & \\ \qquad\qquad \texttt{nil} & \texttt{=>} & e_2\sigma \\ \qquad\qquad x : xs & \texttt{=>} & e_3\sigma \end{array}\right)\right\rangle^H = \left\langle \left(\begin{array}{lrl} \texttt{case}\; e_1\sigma'\; \texttt{of} & & \\ \qquad\qquad \texttt{nil} & \texttt{=>} & e_2\sigma' \\ \qquad\qquad x : xs & \texttt{=>} & e_3\sigma' \end{array}\right)\right\rangle^H \qquad (4.15)$$

Since $e_1\sigma \xrightarrow{H} e_h : e_t$ (by a sub-proof), we have, by an instance of the induction hypothesis that, for some e'_h, e'_t

$$e_1\sigma' \ \xrightarrow{H}\ e'_h : e'_t \ \ \textit{where}\ (e_h : e_t\ \Re^{\cdot}\ e'_h : e'_t) \qquad (4.16)$$
$$\langle e_1\sigma\rangle^H \ =\ \langle e_1\sigma'\rangle^H \qquad (4.17)$$

By the definition of $\Re^{\cdot}$ we have $e_h\ \Re\ e'_h$, and by definition of $\Re$ that for some e_4 containing at most variables $\tilde{y}$

$$e_h \equiv e_4\{\tilde{e}_\alpha/\tilde{y}\} \ \ e'_h \equiv e_4\{\tilde{e}'_\alpha/\tilde{y}\}$$

where $\tilde{e}_\alpha \preceq \tilde{e}'_\alpha$.

Similarly we have some e_5 containing at most variables $\tilde{z}$, and some $\tilde{e}_\beta$, $\tilde{e}'_\beta$ such that $\tilde{e}_\beta \preceq \tilde{e}'_\beta$ and

$$e_t \equiv e_5\{\tilde{e}_\beta/\tilde{z}\} \ \ e'_t \equiv e_5\{\tilde{e}'_\beta/\tilde{z}\}$$

Since we can rename variables, assume *w.l.o.g.* that variables $\tilde{x}$, $\tilde{y}$, $\tilde{z}$, x and xs are all distinct. Using this, and the fact that expressions $\tilde{e}$, $\tilde{e}_\alpha$, $\tilde{e}_\beta$ are closed, we have

$$\begin{aligned}(e_3\sigma)\{e_h/x, e_t/xs\} &\equiv (e_3\sigma)\{(e_4\{\tilde{e}_\alpha/\tilde{y}\})/x, (e_5\{\tilde{e}_\beta/\tilde{z}\})/xs\} \\ &\equiv ((e_3\sigma)\{e_4/x, e_5/xs\})\{\tilde{e}_\alpha/\tilde{y}, \tilde{e}_\beta/\tilde{z}\} \\ &\equiv (e_3\{e_4/x, e_5/xs\})\{\tilde{e}/\tilde{x}, \tilde{e}_\alpha/\tilde{y}, \tilde{e}_\beta/\tilde{z}\}\end{aligned}$$

Similarly we can show

$$(e_3\sigma')\{e'_h/x, e'_t/xs\} \equiv (e_3\{e_4/x, e_5/xs\})\{\tilde{e}'/\tilde{x}, \tilde{e}'_\alpha/\tilde{y}, \tilde{e}'_\beta/\tilde{z}\}$$

Now we have that, since

$$e_3\sigma\{e_h/x, e_t/xs\} \equiv (e_3\{e_4/x, e_5/xs\})\{\tilde{e}/\tilde{x}, \tilde{e}_\alpha/\tilde{y}, \tilde{e}_\beta/\tilde{z}\} \xrightarrow{H} h$$

as an immediate sub-proof, the induction hypothesis gives,

$$\begin{aligned}&(e_3\{e_4/x, e_5/xs\})\{\tilde{e}'/\tilde{x}, \tilde{e}'_\alpha/\tilde{y}, \tilde{e}'_\beta/\tilde{z}\} \xrightarrow{H} h' \\ &\qquad \textit{where}\ \ (h \ \dot{\mathfrak{R}}\ h') \qquad (4.18)\\ &\langle(e_3\{e_4/x, e_5/xs\})\{\tilde{e}/\tilde{x}, \tilde{e}_\alpha/\tilde{y}, \tilde{e}_\beta/\tilde{z}\}\rangle^H \\ &\quad = \left\langle(e_3\{e_4/x, e_5/xs\})\{\tilde{e}'/\tilde{x}, \tilde{e}'_\alpha/\tilde{y}, \tilde{e}'_\beta/\tilde{z}\}\right\rangle^H \qquad (4.19)\end{aligned}$$

From 4.16 and 4.18 we can conclude 4.14 by an application of rule **Case.cons**, and from the definition of $\langle\rangle^H$ together with 4.17 and 4.19 we conclude 4.15 as required.

Case $(e \equiv f(e_1, \ldots e_n))$ requires similar (but simpler) manipulation of substitutions in order to apply the inductive hypothesis.

Cases ($e \equiv$ `if` e_1 `then` e_2 `else` e_3) and $(e \equiv p(e_1, \ldots e_n))$ require a simple application of lemma 4.6.8, but otherwise follow in a routine manner. The remaining cases are straightforward. □

Now we can define our notion of cost-equivalence to be the equivalence relation:

DEFINITION **4.6.14 (cost-equivalence)** $(=_c) \equiv (\preceq \cap \preceq^{-1})$, *i.e.*

$$(e_1 =_c e_2) \iff (e_1 \preceq e_2)\ \&\ (e_2 \preceq e_1)$$

□

So two expressions are *cost-equivalent* if they cost-simulate each other. In fact, for the purposes of reasoning about cost, it will often be sufficient to use the simulation relation.

A context (an expression containing a "hole") can be viewed as an expression containing a single free variable.

COROLLARY **4.6.15** *For all contexts $C[\,]$, if $e =_c e'$, then*

$$\langle C[e]\rangle^\alpha = \langle C[e']\rangle^\alpha$$

PROOF $\alpha = H$, immediate from theorem 4.6.11 and the definition of $\preceq$; $\alpha = N$, immediate from theorem 4.6.11 and lemma 4.6.10 □

Some example cost-equivalence laws can now be given:

PROPOSITION **4.6.16**

$$\begin{array}{lll}
(i) & p(v_1,\ldots,v_n) =_c & v \textit{ if } \mathbf{Apply}(p, v1,\ldots,v_n) = v \\
(ii) & (e_1 + e_2) + e_3 =_c & e_1 + (e_2 + e_3) \\
(iii) & \multicolumn{2}{l}{\texttt{if (if } e_0 \texttt{ then } e_1 \texttt{ else } e_2\texttt{) then } e_3 \texttt{ else } e_4} \\
 & =_c & \texttt{if } e_0 \texttt{ then (if } e_1 \texttt{ then } e_3 \texttt{ else } e_4\texttt{)} \\
 & & \texttt{else (if } e_2 \texttt{ then } e_3 \texttt{ else } e_4\texttt{)}
\end{array}$$

Part (i) is straightforward to prove. Part (ii) is an example of a cost-equivalence inherited from the properties of the primitive functions, and like (iii) can be proved directly by exhibiting the appropriate cost-simulations.

Proposition 4.6.1 given at the start of this section follows easily from part (i) and 4.6.15—it is clear that some of the most useful "axioms" are the laws for the primitive functions (such as (ii)) which also turn out to be cost-equivalences.

4.7 Discussion

The calculus developed directly from the operational semantics in the preceding sections gives a basis for the time analysis of closed expressions. To strengthen this basis we have introduced the notion of cost-simulation which is shown to be a precongruence. This approach has the advantage of relative simplicity, when compared with the translation-method sketched in the previous chapter. In the remainder of this section we discuss the advantages and limitations of this approach.

Equational Reasoning The rules attempt to minimise the amount of operational details required to reason about time-complexity, but we no longer enjoy the equational properties of the language. Establishing an extensional equivalence between two expressions does not allow us to replace one expression by the other in the context of the time equations, since the equations are determined by the *syntax* of expressions. The notion of cost-simulation was introduced to give a substitutive equivalence (with respect to $\langle\ \rangle^H$ and $\langle\ \rangle^N$). Experience is likely to determine a "good" set of cost-equivalence laws—to develop algebraic theories for this equivalence that are as rich as equational theories for the underlying language would require much extra work.

Operational Details The crucial conditional and case expressions require us to reason about expressions of the form $e \xrightarrow{N} v$ and $e \xrightarrow{H} h$; that is to say, the operational semantics is embedded in the time-equations. If we can establish an extensional equivalence $e = v$ then we must have $e \xrightarrow{N} v$. However, it is less straightforward to reason about sentences of the form $e \xrightarrow{H} h$ since establishing the equivalence $e = e_1 : e_2$ does not imply $e \xrightarrow{H} e_1 : e_2$, so in general we must still resort to reasoning directly from the semantic rules to describe properties of the equations defining time-cost.

Extending these ideas to richer languages including higher-order functions is relatively straightforward, although an appropriate cost-simulation will be more complicated—Abramsky's *applicative bisimulation* [Abr90b] gives a suitable starting-point. Modelling lazyness (*i.e.* graph reduction) is possible, but would complicate the operational model and consequently increase the extent to which operational details would be both necessary and undesirable in the derived equations for cost.

Compositionality This approach only allows us to reason about the cost of closed expressions, at best allowing us to express the property in terms of the size-metric of some *value*. Complexity properties of individual functions cannot be assessed in isolation from their context by "operational reasoning" and so we do not have the useful compositional property which allows us to reason about large programs by the study of smaller sub-programs. Unlike call-by-value languages, compositionality is not easily achieved via the direct operational route, since the cost of a function depends on the context in which it is called.

In the next chapter we consider the compositional analysis of a *call-by-need* (*i.e.* lazy) language, where the approach to compositionality goes hand in hand with an indirect description of lazy evaluation.

Chapter 5

Lazy Time Analysis

5.1 Compositionality Through Descriptions of Context

This chapter addresses the problem of giving a compositional time analysis for a lazy language. In chapter 2 we saw how to reason about an expression of the form $f(exp)$ by constructing a sum of

- the cost due to expression *exp*
- an expression denoting the cost due to the application of the function `f` to the value of *exp*.

This division arises naturally in the time-analysis of (first-order) call-by-value programs because the reductions due to expression *exp* occur separately from those due to the application of the function. In the case of a non-strict language this division is not immediate because the function-calls in evaluating *exp* are interleaved with those in the enclosing context f, and cannot be accounted-for independently of f in a straightforward manner. Suppose, for example, that *exp* computes the first fifty prime-numbers, then the cost "due to" *exp* when f is `hd` will be significantly different to when f is `sum`. More generally, we can see that it is the amount of expression *exp* that is *needed* by the function f that determines the cost due to *exp*.

The idea of compositional time-analysis is to parameterise the cost of computing an expression *exp* by a description of the amount of the result that is *needed* by the context in which *exp* appears. This idea is due to Bror Bjerner, whose Ph.D. thesis [Bje89] presents a compositional theory for time analysis of the programs of Martin-Löf type-theory. Programs in the language of Martin-Löf type-theory are primitive-recursive functions with a call-by-name operational semantics. However, a compositional theory of time analysis is best suited to programs with a *lazy* or call-by-need semantics; since an expression may be bound to a variable which occurs multiply (in the right-hand side of some defining equation), such a description of context must be the *net* of each of these separate contexts. In this sense a time analysis based on this "net" description will model lazy (*i.e.* shared) evaluation, since only a single cost is associated with the expression.

In [WH87], Wadler and Hughes introduced a new kind of strictness analysis[1] based on domain *projections*. One interesting contribution of this approach (which is also shared by some other approaches, *e.g.* [Wra86, Hug87]) is that it can determine in some instances when a parameter is *not needed*[2].

It is this characterisation of "not-need" based on the projection approach that enabled Wadler [Wad88] to give a simpler account of Bjerner's approach (but applied to first-order functional programs rather than the programs in type-theory), where projections take the place of Bjerner's *evaluation degrees*. In addition to a simplified description of "not-need", the strictness analysis motivation of [WH87] gives a natural notion of *approximation* of context-information which is lacking from Bjerner's calculus, and gives rise to techniques for automatically determining a subset of the instances of not-neededness in a program.

In the first part of this chapter we present a refinement of the approach of [Wad88] based on the observation that *strictness* information tells us that (at least part of) a function's argument is needed. Using a certain class of projections we introduce two types of time-equation:

- *sufficient-time* equations (corresponding to the equations in [Wad88]) which use information about "not-neededness" to give an upper-bound to the time for lazy evaluation.

- *necessary-time* equations, a dual lower-bound, with better safety properties, which use information about neededness (strictness).

The chapter is organised as follows. Firstly we introduce projections as a description of *context* that will be used in the analysis of lazy languages. We use this to develop *sufficient-time* analysis, an upper-bound analysis for a lazy first-order language, which uses contexts that describe information that is *sufficient* to com-

[1]In its simplest form, a function f is strict if $f(\perp) = \perp$

[2]If, for all x, $f(x) = f(\perp)$ then we can say that function f does not need its argument

pute a value. We then present *necessary-time* analysis, a corresponding lower-bound analysis.

The next step extends these ideas to a higher-order language using the techniques of chapter 3.

5.2 Modelling Contexts with Projections

In this section we provide a introduction to the use of *projections* as a description of *context*. The formulation and notation will be closest to that provided in [WH87] (where the the application is to strictness analysis). A more technically detailed account (plus some extensions) is considered in [DW89]. Some alternative notation for this work can be found in [Bur90].

Since we are basing our time analysis on the notion of projections from [WH87], we are implicitly restricting our discourse to strongly-typeable expressions—although this extends to parametric polymorphism [Hug88].

The basic problem is, given a function, how much information do we require from the argument in order to determine a certain amount of information about the result. A projection, in the domain theoretic sense, can provide a concise description of both the amount of information which is *sufficient* and, by a slight trick, the amount which is *necessary*. So, what is a projection?

DEFINITION **5.2.1** *A projection, α is a continuous function from a domain $\mathcal{D}$ onto itself, such that*

$$\begin{aligned} \alpha &\sqsubseteq \text{ID}_{\mathcal{D}} \\ \alpha \circ \alpha &= \alpha \end{aligned}$$

where $\text{ID}_{\mathcal{D}}$ is the identity function on $\mathcal{D}$

□

In other words, given an object u, a projection removes information from that object ($\alpha\, u \sqsubseteq u$), but once this information has been removed further application has no effect ($\alpha(\alpha\, u) = \alpha\, u$).

A projection will give us a partial description of a context, *viz.* the information removed from an object by the application of the projection is a description of information *not needed* by the context. In the extreme case, a context may not require any information from an expression. In this case the appropriate projection is BOT which throws away all information:

$$\text{For all } u,\ \text{BOT}\, u = \bot$$

Suppose we have some context $C[\,]$. Then suppose a projection α is the denotation of some expression A. Then we could say that α is a description of context $C[\,]$ if for all expressions e, $C[e] = C[A(e)]$

Generalising this idea we focus on *functions* rather than the more syntactic notions of contexts (expressions with "holes"). Suppose then that we have a (first order) function,

$$f : D_1 \times \cdots \times D_n \rightarrow D$$

(denoting, say, some user-defined function). If we have a description (a projection $\alpha : D \rightarrow D$) of the amount of the result of an application of f that is not needed, then the natural question to ask is: how much of f's arguments are needed? We can describe this by a projection ($\beta : D_i \rightarrow D_i$) on the argument, and the relation between projections α and β is that they must satisfy the *safety condition.*

Definition **5.2.2 The Safety Condition** *If, given some function f, and projections α and β of the appropriate type,*

$$\alpha(f(u_1,\ldots,u_n)) = \alpha(f(u_1,\ldots,(\beta u_i),\ldots u_n))$$

for all objects $u_1,\ldots,u_n$, then we say that in an α-context, f is β-safe in its i^{th} argument.

□

This is abbreviated in the notation of [WH87] by

$$f^i : \alpha \Rightarrow \beta\ .$$

Now supposing $f^i : \alpha \Rightarrow \beta$ for some α, β. Then for all β' such that $\beta \sqsubseteq \beta'$, $f^i : \alpha \Rightarrow \beta'$. So we see that the safety condition says nothing about preciseness of context descriptions, but a smaller projection on a function argument (satisfying the safety condition) is a more precise description of the amount of the argument needed. More informally, we will describe a projection α as being *safe* for some expression e, meaning that we can replace e by αe in some implicit program context, without changing the meaning of the program.

5.2.1 Projections for Strictness

We will require that projections describe two types of information: What information is *sufficient*, and what information is *necessary.* In order to describe the latter, (in order to express when a function is strict), Wadler and Hughes introduce a new domain element, $\lightning$, called "abort". The interpretation of $\alpha u = \lightning$ is that context α requires a value more defined than u. If $\alpha u \neq \lightning$ then we will also say that u *satisfies* α.

To make this trick work, we must have $\lightning \sqsubset \perp$ and all functions are naturally extended to be strict in $\lightning$, *i.e.*,

$$f(u_1, \ldots, \lightning, \ldots, u_n) = \lightning$$

These technical devices are explained more formally in terms of *lifting* of the domains: more formal notation is used in [DW89, Bur90], but we will use the informal use of an "abort" element below bottom, and we will implicitly assume that all domains $\mathcal{D}$ are lifted (written $\mathcal{D}_{\lightning}$).

These devices allow for the definition of an important class of projections which we shall call *lift-strict*[3]:

DEFINITION **5.2.3** *A lift-strict projection is any projection α such that $\alpha(\perp) = \lightning$*

□

These are the projections that require a value to be more defined than $\perp$. The largest of such projections is STR:

$$\textsc{str}(u) = \begin{cases} \lightning & \text{if } u = \perp \text{ or } u = \lightning \\ u & \text{otherwise} \end{cases}$$

and this allows us to describe ordinary strictness since ([WH87])

$$f\perp = \perp \iff \textsc{str} \circ f \circ \textsc{str} = \textsc{str} \circ f$$

As an equivalent alternative definition of lift-strict projections: a projection α is lift-strict if and only if $\alpha \sqsubseteq \textsc{str}$.

5.2.2 The Projection ABS

Of the non-lift-strict projections, the smallest is the projection ABS, the lifted version of BOT:

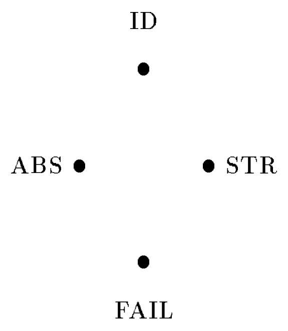
$$\textsc{abs}(u) = \begin{cases} \lightning & \text{if } u = \lightning \\ \perp & \text{otherwise} \end{cases}$$

This context is important since if it is safe to evaluate an expression in the context ABS, then the value of the expression will not be needed.

[3]This terminology is from [Bur90]—they are simply called *strict projections* in [WH87]

5.2.3 The Projection Lattice

A projection $\alpha : \mathcal{D}_{\lightning} \rightarrow \mathcal{D}_{\lightning}$, will be called *a projection over* $\mathcal{D}$. Projections over any domain form a lattice, with ordering $\sqsubseteq$, containing the following points:



where FAIL is the unsatisfiable context FAIL $u = \lightning$ for all u. *i.e.* it requires "the impossible" of a value (*e.g.* that it is simultaneously a non-empty and an empty list) in order that the program is well-defined. There are potentially infinitely many projections on a given non-finite domain, each of which is either lift-strict (FAIL $\sqsubseteq \alpha \sqsubseteq$ STR) or non-lift-strict (ABS $\sqsubseteq \alpha \sqsubseteq$ ID). The four projections above will be used "polymorphically" to represent the corresponding projection over the appropriate domain.

5.2.4 List Projections

Since we will give some examples of functions over lists, we introduce some useful devices for building objects denoting projections over the non-flat domain of lists of elements from some domain:

NIL is the context which requires an empty-list

$$\text{NIL}\, u = \begin{cases} nil & \text{if } u = nil \\ \lightning & \text{otherwise} \end{cases}$$

It is also useful to define a "context-constructor" CONS

$$\text{CONS}\, \alpha\, \beta\, u = \begin{cases} cons(\alpha\, x)(\beta\, xs) & \text{if } u = cons\ x\ xs \\ \lightning & \text{otherwise} \end{cases}$$

So CONS $\alpha\,\beta$ is the context which requires a non-empty list whose head is needed in context α and whose tail is needed in context β. For example, the context (CONS STR ABS) requires a non-empty list whose first element is needed, and whose remaining elements are not.

5.2.5 Determining Safe Projections

In order to give a compositional time analysis we will require projections satisfying the safety condition for the user-defined functions (as well as the primitive functions) of the program. Analysing context is naturally expressed as a *backwards analysis* [AH87]: given a projection α for a function f, what can we say about the projections of the arguments? We need to propagate the information about the result of a function *backwards* to it's arguments. *i.e.*, given a function f of arity n, and a context α we need to find β_i such that

$$\alpha(f(u_1,\ldots,u_n)) \sqsubseteq f(u_1,\ldots,(\beta_i u_i),\ldots,u_n)$$

for all objects $u_1 \ldots u_n$. Ideally we need to find the smallest β, since this describes the context most precisely. However, even though smallest β exist for the stable functions [Hun90b], they are not computable, and so we have two options:

- Use an automatic method for determining some projections satisfying the safety condition.
- Work within a more general but non-mechanisable calculus for reasoning about projections satisfying the safety condition.

Our emphasis here is on the use of information given by projections for functions that satisfy the safety condition, rather than the methods for determining such projections. Fully mechanisable techniques for determining safe projections for functions are given in [WH87]: the method here (as in many program analyses—see [AH87] for examples) is to choose a finite lattice for projections on a given domain. Limitations of this approach and more general techniques are discussed in section 5.6.2.

For the present we assume we have some appropriate projection information in the form of *projection transformers*: mappings from a projection describing the output of a function to a projection describing an argument.

Projection Transformers

Given some α, a function which returns a β_i which satisfies

$$\alpha(f(u_1,\ldots,u_n)) = \alpha(f(u_1,\ldots,(\beta_i u_i),\ldots u_n))$$

is called a *projection transformer* for f ([WH87]). We will adopt the following notation: given a function definition of the form

$$f(x_1,\ldots,x_n) = e$$

the projection transformer denoted by $f^{\#i}$ satisfies

$$\alpha(f(u_1,\ldots,u_n)) \sqsubseteq (f(u_1,\ldots,(\beta_i u_i),\ldots,u_n)), \text{ where } f^{\#i}\alpha = \beta_i$$

N.B. Strictly speaking we should distinguish between the syntactic objects—the program defining f, and the semantic objects—projections, and the denotations given by some semantic function. Following the style of [WH87] we will mix these entities for notational convenience.

As examples, consider the projection transformers for the primitive functions `hd` and `tl`:

$$\begin{aligned} \mathtt{hd}^{\#1}\alpha &= \text{CONS}\ \alpha\ \text{ABS} \\ \mathtt{tl}^{\#1}\alpha &= \text{CONS ABS}\ \alpha \end{aligned}$$

The intuitive reading for $\mathtt{hd}^{\#1}$ is as follows: if we require α's worth of the result of applying `hd`, then we need its argument to be a cons-cell, whose head "satisfies" α but whose tail is not needed. The projection transformers for `cons` satisfy the equations

$$\begin{aligned} \mathtt{cons}^{\#1}(\text{CONS}\ \alpha\ \beta) &= \alpha \\ \mathtt{cons}^{\#2}(\text{CONS}\ \alpha\ \beta) &= \beta \end{aligned}$$

5.3 Sufficient-Time Analysis

In this section we show how context information can be used to aid the time analysis of a lazy first-order language; sufficient-time analysis (with some minor differences) corresponds to the time analysis presented in [Wad88]. The information obtained by the backwards analysis is used to derive equations which compute an upper bound to the precise cost of a given program. This upper-bound is obtained by using information which tells us what values are *sufficient* to compute an expression. We call the resulting analysis a *sufficient-time* analysis.

5.3.1 Context-Parameterised Cost-Functions

As in the first-order eager time analysis, we will define a *cost-function*, cf_i, for each function f_i defined in the original program. As before the cost functions will take as parameters the original arguments to the functions, but in addition they will be parameterised by a *context*, representing the context in which the functions are evaluated.

How can cost-functions make use of context?

We know that any expression in the context ABS will be ignored, so the cost in this context is zero. In any other context the cost of a function application will be (approximated above by) 1 (since we are counting the reduction of function applications) + the cost of evaluating the body of the function, in that context.

We define the cost functions associated with each function

$$f_i(x_1 \ldots x_{n_i}) = e_i$$

to be

$$cf_i(x_1, \ldots, x_{n_i}, \alpha) = \alpha \hookrightarrow_s 1 + \mathcal{T}_s[\![e_i]\!]\alpha$$

Where we introduce the notation $\alpha \hookrightarrow_s e$ to abbreviate cost e "guarded" by context α:

$$\alpha \hookrightarrow_s e = \begin{cases} 0 & \text{if } \alpha = \text{ABS} \\ e & \text{otherwise} \end{cases}$$

The syntactic map $\mathcal{T}_s[\![\ ]\!]$ defined in figure 5.1 is very similar to that defined in chapter 2 (figure 2.4), but is defined with respect to a particular context. $\mathcal{T}_s[\![e]\!]\ \alpha$ defines the cost of evaluating expression e in context α. It makes use of the context transformers

$$f_i^{\#1} \ldots f_i^{\#n_i}$$

defined for each function f_i, which satisfy the required safety criterion (and could be defined according to the backwards analysis of [WH87]). In particular it will be appropriate to set $f^{\#i}(\text{ABS}) = \text{ABS}$ since if the result of a function is not needed, then neither are its arguments.

PROPOSITION **5.3.1** *for all functions f, all satisfiable contexts β (i.e., $\beta \neq$ FAIL) and objects $u_1 \ldots u_n$,*

$$\text{ABS}(f(u_1, \ldots, u_n)) = \text{ABS}(f(u_1, \ldots, \beta(u_i), \ldots, u_n)) \iff \text{ABS} \sqsubseteq \beta$$

PROOF

($\Rightarrow$) Suppose $\text{ABS}(f(u_1, \ldots, u_n)) = \text{ABS}(f(u_1, \ldots, \beta(u_i), \ldots, u_n)) = \bot$. Then we must have $f(u_1, \ldots, \beta(u_i), \ldots, u_n) \neq \lightning$. Taking $u_i = \bot$, we must have β non-lift-strict, *i.e.* $\text{ABS} \sqsubseteq \beta$.

$$
\begin{aligned}
\mathcal{T}_s[\![c]\!]\alpha &= 0\\
\mathcal{T}_s[\![x]\!]\alpha &= 0\\
\mathcal{T}_s[\![\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3]\!]\alpha &= \alpha \hookrightarrow_s\\
&\quad \mathcal{T}_s[\![e_1]\!]\textsc{id}\\
&\quad + \texttt{if } e_1 \texttt{ then } \mathcal{T}_s[\![e_2]\!]\alpha \texttt{ else } \mathcal{T}_s[\![e_3]\!]\alpha\\
\mathcal{T}_s[\![p(e_1 \dots e_n)]\!]\alpha &= \mathcal{T}_s[\![e_1]\!](p^{\#1}\alpha) + \cdots + \mathcal{T}_s[\![e_n]\!](p^{\#n}\alpha)\\
\mathcal{T}_s[\![f_i(e_1 \dots e_n)]\!]\alpha &= cf_i(e_1 \dots e_n, \alpha)\\
&\quad + \mathcal{T}_s[\![e_1]\!]({f_i}^{\#1}\alpha) + \cdots + \mathcal{T}_s[\![e_n]\!]({f_i}^{\#n}\alpha)
\end{aligned}
$$

Figure 5.1: Definition of $\mathcal{T}_s[\![\ ]\!]$

($\Leftarrow$) When $\textsc{abs}(f(u_1, \dots, u_n)) = \lightning$ then $f(u_1, \dots, u_n) = \lightning$ and so, since β is a projection (and f is monotonic)

$$f(u_1, \dots, \beta(u_i), \dots, u_n) = \textsc{abs}(f(u_1, \dots, \beta(u_i), \dots, u_n)) = \lightning$$

When $\textsc{abs}(f(u_1, \dots, u_n)) = \bot$ then $f(u_1, \dots, u_n) \neq \lightning$ and so $u_j \neq \lightning$, $j = 1 \dots n$, so $\beta(u_i) \sqsupseteq \textsc{abs}(u_i) = \bot$. So $f(u_1, \dots, \beta(u_i), \dots, u_n) \neq \lightning$ which implies

$$\textsc{abs}(f(u_1, \dots, u_n)) = \bot$$

$\Box$

Function Application

The rule for function application embodies the idea of the backwards propagation of a context by the projection transformers. This rule tells us that the cost of evaluating a function application is the associated cost-function applied to the arguments (and the context) plus the sum of evaluating the arguments in the contexts prescribed by the context-transformers.

Conditional

The conditional expression, like any other, has zero cost in the context ABS (hence the use of the $\hookrightarrow_s$ operator). Otherwise we sum the cost of evaluating the condition (which may or may not be evaluated, hence the safe-context for boolean values ID) plus either the cost of the alternate or the consequent, depending on the *value* of the condition.

A Small Example

Consider the following program

$$\texttt{hd (cons(not(true),}\mathit{exp}\texttt{))}$$

where

```
not(x) = if x then false else true
```

and *exp* represents some arbitrary expression. The cost-function for `not` is

$$\begin{aligned}\texttt{cnot(x,}\alpha\texttt{)} &= \alpha \hookrightarrow_s \texttt{1 + (} \alpha \hookrightarrow_s \texttt{0 + if x then 0 else 0)}\\ &= \alpha \hookrightarrow_s \texttt{1}\end{aligned}$$

We assume a boolean-valued program is evaluated in the context STR, and so the cost program is defined by:

$$\mathcal{T}_s[\![\texttt{hd(cons(not(true),}\mathit{exp}\texttt{))}]\!]\ \text{STR}$$

Which is, by definition

$$\begin{aligned}&\texttt{cnot(true , cons}^{\#1}\texttt{(hd}^{\#1}\texttt{(}\text{STR}\texttt{))) + 0 +}\\ &\mathcal{T}_s[\![\mathit{exp}]\!]\ \texttt{cons}^{\#2}\texttt{(hd}^{\#1}\texttt{(}\text{STR}\texttt{))}\end{aligned}$$

The context transformers for the primitive functions satisfy

$$\begin{aligned}\texttt{hd}^{\#1}(\alpha) &= \text{CONS}\ \alpha\ \text{ABS}\\ \texttt{cons}^{\#1}(\text{CONS}\ \alpha\ \beta) &= \alpha\\ \texttt{cons}^{\#2}(\text{CONS}\ \alpha\ \beta) &= \beta\end{aligned}$$

and so the cost is

$$\texttt{cnot(true , }\text{STR}\texttt{)} + \mathcal{T}_s[\![\mathit{exp}]\!]\ \text{ABS} = 1$$

for any expression *exp*.

5.3.2 Correctness

What are the precise properties of the cost programs? Here we consider the approximation and correctness properties of the "lazy" cost-program.

Approximation

We wish to show that expression $\mathcal{T}_s[\![e]\!]\alpha$ gives an upper-bound estimate to the cost of lazy evaluation of e in context α. At first sight it would seem that the very minimum we require for this task is an explicit lazy operational-semantics. This would require, for example, a semantics based on the manipulation of graphs (*e.g.* [BvEG+87]) or a Plotkin-style semantics involving a (side-effected) *store*. Whatever approach we choose, we would certainly have to deal with a semantics rather more complicated (and more difficult to reason about) than those we have encountered so far.

We will give an informal correctness proof which *avoids* the use of a direct operational model of lazy evaluation. This is possible by exploiting:

- results relating the numbers of reduction steps using call-by-value and call-by-need
- correctness of the call-by-value scheme of chapter 2
- relations between the call-by-value scheme and the sufficient-time scheme
- simple operational properties implied by safe projection information

A connection between the number of reductions in call-by-value and in a lazy semantics seems fairly well-known in functional programming:

> The number of reduction-steps for evaluation to normal-form using lazy (call-by-need) evaluation is less than or equal to the number required using call-by-value.

However, a general proof of this property is rather less well known; Wadsworth ([Wad71] chapter 4), who introduced the call-by-need parameter mechanism in the λ-calculus suggested that this property held, but was unable to prove it. This assertion remained unproved in [HM76] where the "lazy cons" was introduced. This property was proved, at least for first-order recursive program schemes (which is sufficient here), in [O'D77] (Theorem 15, p65).

The correctness of the cost-function scheme $\mathcal{T}[\![\cdot]\!]$ given in chapter 2 implies, by the above property, that the call-by-value cost-functions give an upper-bound to the cost of lazy evaluation.

The sufficient-time for evaluating an expression in context α is less than the time under call-by-value.

THEOREM **5.3.2** *For all expressions e, and contexts α, $\mathcal{T}_s[\![e]\!]\alpha \leq \mathcal{T}[\![e]\!]$, whenever both sides of the inequality are well-defined.*

PROOF (Sketch)

Apart from the additional parameterisation of cost-functions by a projection, and the propagation of this projection by some projection-transformers, the only syntactic difference in the schemes $\mathcal{T}_s[\![\ ]\!]$ and $\mathcal{T}$ are the uses of the $\hookrightarrow_s$ guards, which serve to "filter" certain cost-expressions to give a smaller cost than in the call-by-value schemes. □

A more formal proof can be given as either a fixed-point (Scott) induction on the denotations of the cost-functions, or an induction on the structure of the proof in an operational semantics—however, the exact choice of semantics (strict or non-strict) for the *cost-functions* is not central to the theorem.

We know that $\mathcal{T}[\![e]\!]$ is an upper-bound, but to show that $\mathcal{T}_s[\![e]\!]\alpha$ is also an upper-bound we need to show that the "filtering" of cost-expressions by $\hookrightarrow_s$ is sound. Since the only cost-expressions that are filtered are those corresponding to expressions evaluated in context ABS, we require the following:

THEOREM **5.3.3** *If* ABS *is a safe context-description for some expression e in some terminating program, then the cost due to e is zero.*

PROOF (Sketch)

If α is a safe context for e in some program, then we can replace e by $\alpha(e)$ without changing the meaning of the program. With $\alpha =$ ABS we can replace e by ABS$(e) = \bot$. This implies that the expression e must never get evaluated, since if an attempt was made to evaluate it, the program would not terminate. □

Together with the assumption that the projection-transformers always give a safe context, we then conclude that sufficient-time equations, when well-defined, give an upper-bound to lazy evaluation time.

Furthermore, we can show that the definition $\mathcal{T}_s[\![\,]\!]$ *always* gives the appropriate result with respect to ABS; namely, the sufficient-time for evaluating any expression in the context ABS is zero:

PROPOSITION **5.3.4** *For every expression e,* $\mathcal{T}_s[\![e]\!]$ ABS $= 0$

PROOF Structural induction in e:

- *constants, identifiers* : Immediate from definition of $\mathcal{T}_s[\![\,]\!]$
- *conditional* : Immediate from definition of $\hookrightarrow_s$

- *primitive functions* :

$$\begin{aligned}
\mathcal{T}_s[\![p(e_1,\ldots e_n,)]\!]\text{ABS} &= \mathcal{T}_s[\![e_1]\!](p^{\#1}\text{ABS}) + \cdots + \mathcal{T}_s[\![e_n]\!](p^{\#n}\text{ABS}) \\
&= \mathcal{T}_s[\![e_1]\!]\text{ABS} + \cdots + \mathcal{T}_s[\![e_n]\!]\text{ABS} \\
&\quad \text{– by proposition 5.3.1} \\
&= 0 + \cdots + 0 \\
&\quad \text{– by inductive hypothesis}
\end{aligned}$$

- *non-primitive functions* :

$$\begin{aligned}
\mathcal{T}_s[\![f(e_1,\ldots,e_n)]\!]\text{ABS} &= \mathcal{T}_s[\![e_1]\!](f^{\#1}\text{ABS}) + \cdots + \mathcal{T}_s[\![e_n]\!](f^{\#n}\text{ABS}) \\
&\quad + cf(e_1,\ldots,e_n,\text{ABS}) \\
&= \mathcal{T}_s[\![e_1]\!]\text{ABS} + \cdots + \mathcal{T}_s[\![e_n]\!]\text{ABS} + 0 \\
&\quad \text{– by proposition 5.3.1 and } \hookrightarrow_s \\
&= 0 + \cdots + 0 \\
&\quad \text{– by inductive hypothesis}
\end{aligned}$$

□

Since the safety condition for projections does not specify that we require the smallest possible projection, the context ABS may be approximated by any larger projection.

This approximation is reflected in the cost-program as an over-estimation of cost. (In the extreme case the context transformers are such that the context ABS is *never* derived in the cost-program, and so the value of the cost program is the same as that given by the strict derivation of figure 2.4, page 28.)

To see why we have an upper-bound on lazy-evaluation, rather than just call-by-name we give the following intuition (in addition to theorem 5.3.2): in computing the cost of a function application $f(e)$ in context α the cost due to e will only be counted *once*. The context of e, $f^{\#1}(\alpha)$ will be the *net* context of the possible contexts in which e is shared.

Safety

So far we have glossed-over a problem with this analysis method; whenever the cost-program terminates yielding a value, that value is indeed an upper-bound to the time cost of evaluating the program lazily. However, there are cases when the cost-program does not yield a value when it should do so. These fall into two classes:

(i) *Nontermination*: The cost-program may not terminate even when the program does. This occurs, for example, when the cost of an uncomputed expression is counted due to the approximation in the analysis. Consider the following simple example to illustrate this:

```
loop(b)        =  loop(b)
alwaystrue(x)  =  if false then x else true
-------------------------------------------
alwaystrue(loop(false))
```

The cost program derived is

```
cloop(x,α)        =  α ↪s 1 + cloop(x,α)
calwaystrue(x,α)  =  α ↪s 1 + if false then 0 else 0
-----------------------------------------------------
calwaystrue(loop(false),STR) +
cloop(false, alwaystrue#(STR))
```

Although it is easy in this example to show that `alwaystrue`$^{\#}$(STR) = ABS, which gives a terminating cost-program, a mechanised analysis based on [WH87] would yield the "safe" result `alwaystrue`$^{\#}$(STR) = ID using the four point context domain for truth-values. This tells us that `alwaystrue` *may or may not* require its value in a strict context. The cost estimate in this case would then be

```
   calwaystrue(loop(false),STR) + cloop(false,ID))
=  1 + cloop(false,ID))
=  ...
=  1 + 1 + cloop(false,ID)
=  ...
```

Non-terminating cost expressions of this kind can be thought of as "computing" the worst possible upper-bound to the cost.

(ii) *Run-time errors:* The approximation in the cost-program can lead to an attempt to compute not only the cost, but also the *value* of ignored expressions (even though we are considering lazy cost-expressions). Consider the following function, and its cost-function (with minor simplification)

```
sometimestrue(x)      =  if eq(x,0)
                         then alwaystrue(false)
                         else false


csometimestrue(x,α)  =  α ↪s 1 + if eq(x,0)
                                   then calwaystrue(false,α)
                                   else 0
```

Now consider the cost of the expression

```
alwaystrue(sometimestrue(div(1,0)))
```

in the context STR, where "`div(1,0)`" is intended to be some expression which, if evaluated, causes the program to abort. In this case the cost-expression is

```
     calwaystrue(sometimestrue(div(1,0)),STR)
      + csometimestrue(div(1,0),alwaystrue#(STR))
  =  1 + csometimestrue(div(1,0),alwaystrue#(STR))
```

If, once again, we use the "approximate" context-transformer above, we get

```
1 + csometimestrue(div(1,0),ID)
```

which requires the value of the abortive expression `div(1,0)`.

5.4 Necessary-Time Analysis

We have outlined the use of projections to derive equations which can give an upper-bound to the time-cost of an expression in a particular context. The cost-functions which compute this sufficient-complexity are only partially correct in the sense that if they compute a value, then that value is indeed an upper-bound to the time-cost of a program. There is potentially much more information about contexts using the projections over domains extended to include $\lightning$ (*i.e.* projection lattices containing lift-strict projections, the projections which enable us to describe strictness properties). In fact, in the sufficient-time analysis we can, for example identify the contexts ID and STR since they provide exactly the same information about sufficient cost. (This fact manifests itself in [Wad88] as the use of some unsafe projection transformers which, although they derive the context STR when they should give ID, yield a correct time analysis.)

The lift-strict contexts allow us to describe the amount of information which is *necessary* to compute a value. In this section we show how the use of this information can give us equations which describe a lower-bound on the precise time-cost (the *necessary-time*), and which overcome the termination deficiencies of sufficient-time analysis. The key to sufficient-time analysis is the use of the context ABS to deduce that an expression will not be evaluated. We will introduce a *necessary-time* whose key is the use of lift-strict projections.

5.4.1 Necessary-Cost Functions

In order to construct functions which compute the necessary-cost of evaluating a function in a particular context, we make the following operational connection between expressions which can be safely evaluated in a lift-strict context, and their operational behaviour.

THEOREM **5.4.1** *If, in some program, it is safe to evaluate an expression of the form $f(e_1, \ldots, e_n)$ in a lift-strict context, then operationally if the program is well-defined we know that* at least *one reduction step of f must be performed.*

PROOF Safety implies we can replace $f(e_1, \ldots, e_n)$ with $\alpha(f(e_1, \ldots, e_n))$ for some FAIL $\sqsubset \alpha \sqsubseteq$ STR, without changing the meaning of the program. Thus if the program is well-defined we must have $f(e_1, \ldots, e_n) \sqsupset \bot$. Operationally this implies that $f(e_1, \ldots, e_n)$ must be evaluated to get (at least) some head-normal form. To achieve this, the least we must do is evaluate this application to examine the body of f, which means we perform at least one evaluation step. □

Conversely, if an expression is evaluated in a non-lift-strict context then that expression *may or may not* be reduced (only the context ABS allows us to conclude that it *definitely* will not).

Motivated by this observation, we now define the necessary-cost; the cost of evaluating an expression e in a context α is given by $\mathcal{T}_n[\![e]\!]\alpha$ where $\mathcal{T}_n[\![\;]\!]$ is once again a mapping defined over the syntax of expressions, and assuming some safe context transformers for the user-defined functions.

For each function definition of the form $f_i(x_1 \ldots x_{n_i})$ = e_i we will define an associated necessary-cost function

$$cf_i(x_1, \ldots, x_{n_i}, \alpha) = \alpha \hookrightarrow_n 1 + \mathcal{T}_n[\![e_i]\!]\alpha$$

The notation $\alpha \hookrightarrow_n e$ is used to abbreviate necessary-cost e (depending on α) modulo context α:

$$\alpha \hookrightarrow_n e = \begin{cases} e & \text{if } \alpha \sqsubseteq \textsc{str} \\ 0 & \text{otherwise} \end{cases}$$

The definition of $\mathcal{T}_n[\![\ ]\!]$ is given in figure 5.2. The rules are very similar to the

$$\begin{array}{rcl}
\mathcal{T}_n[\![c]\!]\alpha & = & 0 \\
\mathcal{T}_n[\![x]\!]\alpha & = & 0 \\
\mathcal{T}_n[\![\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3]\!]\alpha & = & \alpha \hookrightarrow_n \\
& & \mathcal{T}_n[\![e_1]\!]\textsc{str} \\
& & + \texttt{ if } e_1 \texttt{ then } \mathcal{T}_n[\![e_2]\!]\alpha \texttt{ else } \mathcal{T}_n[\![e_3]\!]\alpha \\
\mathcal{T}_n[\![p(e_1 \ldots e_n)]\!]\alpha & = & \mathcal{T}_n[\![e_1]\!](p^{\#1}\alpha) + \cdots + \mathcal{T}_n[\![e_n]\!](p^{\#n}\alpha) \\
\mathcal{T}_n[\![f_i(e_1 \ldots e_n)]\!]\alpha & = & cf_i(e_1 \ldots e_n, \alpha) \\
& & + \mathcal{T}_n[\![e_1]\!]({f_i}^{\#1}\alpha) + \cdots + \mathcal{T}_n[\![e_n]\!]({f_i}^{\#n}\alpha)
\end{array}$$

Figure 5.2: Definition of $\mathcal{T}_n[\![\ ]\!]$

definitions for $\mathcal{T}_s[\![\ ]\!]$ where we use the guard $\hookrightarrow_n$ in place of $\hookrightarrow_s$. The only significant difference is in the translation for the conditional expression.

PROPOSITION **5.4.2** *For all contexts α, if $\alpha \sqsubseteq \textsc{str}$ then*

$$\alpha(\texttt{if } u_1 \texttt{ then } u_2 \texttt{ else } u_3) = \alpha(\texttt{if } \textsc{str}(u_1) \texttt{ then } u_2 \texttt{ else } u_3)$$

PROOF Split into two cases according to u_1:

(i) $u_1 \sqsubseteq \bot$: Then we must have $(\texttt{if } u_1 \texttt{ then } u_2 \texttt{ else } u_3) \sqsubseteq \bot$. Since α is lift-strict we have

$$\begin{array}{rcl}
\alpha(\texttt{if } u_1 \texttt{ then } u_2 \texttt{ else } u_3) & = & \lightning \\
& = & \alpha(\texttt{if } \lightning \texttt{ then } u_2 \texttt{ else } u_3) \\
& = & \alpha(\texttt{if } \textsc{str}(u_1) \texttt{ then } u_2 \texttt{ else } u_3)
\end{array}$$

(ii) $u_1 \sqsupset \bot$: Then we have $\textsc{str}(u_1) = u_1$ and the result follows immediately.

□

This tells us that in any strict context it is safe to evaluate the condition in the context STR(*i.e.*, `if` is strict in its first argument), and gives us the appropriate context for determining the cost due to the condition in the conditional expression.

5.4.2 Example

As an example we use insertion-sort function. The equations are given in figure 5.3.

```
insert(x,xs)  =  if null(xs)
                 then cons(x,nil)
                 else if x < hd(xs)
                      then cons(x,xs)
                      else cons(hd(xs),insert(x,tl(xs)))

sort(xs)      =  if null(xs)
                 then nil
                 else insert(hd(xs),sort(tl(xs)))

min(xs)       =  hd(sort(xs))
```

Figure 5.3: Insertion Sort

Necessary time

In this example we wish to consider the cost of evaluating `min` in a strict context. The necessary-time equations constructed according to $\mathcal{T}_n[\![\ ]\!]$ are given in figure 5.4. In this example we wish to consider the cost of evaluating `min` in a strict context. We are not particularly concerned here with the techniques for deriving the safe projection transformers. We note however that the projection transformers needed in this example are (implied by) members of the finite domains for lists (and integers) described in [WH87] for the purpose of strictness analysis, and as such can be determined mechanically by fixpoint iteration. The equations we require are:

$$\begin{aligned}
\texttt{hd}^{\#}(\text{STR}) &= \text{CONS STR ABS} \\
\texttt{cons}^{\#2}(\text{CONS STR ABS}) &= \text{ABS} \\
\texttt{insert}^{\#2}(\text{CONS STR ABS}) &= \text{CONS STR ABS}
\end{aligned}$$

```
cinsert(x,xs,$\alpha$)  =  $\alpha \hookrightarrow_n$
                     1 + if null(xs)
                         then 0
                         else if x < hd(xs)
                              then 0
                              else cinsert(x,tl(xs),cons$^{\#2}$($\alpha$)))
csort(xs,$\alpha$)      =  $\alpha \hookrightarrow_n$
                     1 + if null(xs)
                         then 0
                         else cinsert(hd(xs),sort(tl(xs)),$\alpha$) +
                         csort(tl(xs),insert$^{\#2}$($\alpha$))

cmin(xs,$\alpha$)       =  $\alpha \hookrightarrow_n$1 + csort(xs,hd$^{\#1}$($\alpha$))
```

Figure 5.4: Necessary-Cost Functions

Now we examine the cost of `min`

```
     cmin(xs,STR)
  =  STR $\hookrightarrow_n$1 + csort(xs,hd$^{\#1}$(STR)
  =  1 + csort(xs,CONS STR ABS)
```

```
     csort(xs,CONS STR ABS)
  =  1 + if null(ys)
         then 0
         else cinsert(hd(xs),sort(tl(xs)),CONS STR ABS) +
              csort(tl(xs),insert$^{\#2}$(CONS STR ABS))
  =  1 + if null(xs)
         then 0
         else cinsert(hd(xs),sort(tl(xs)),CONS STR ABS) +
              csort(tl(xs),CONS STR ABS)
     cinsert(y,ys,CONS STR ABS)
  =  1 + if null(ys)
         then 0
         else if y < hd(ys) then 0
              else cinsert(y,tl(ys),cons#2(CONS STR ABS)))
  =  1 + if null(xs)
         then 0
         else if y < hd(ys) then 0
              else cinsert(y,tl(ys),ABS)
  =  1
```

and so

```
     csort(xs,CONS STR ABS)
  =  1 + if null(xs)
         then 0
         else 1 + csort(tl(xs),CONS STR ABS)
```

This simple recurrence has the exact solution `1 + 2*length(xs)` and so

```
cmin(xs,STR) = 2 + 2*length(xs)
```

Sufficient time

In this example the sufficient-time equations look very similar to the necessary-time equations above. The difference is in the use of the strict context guard $\hookrightarrow_n$. However, since the contexts (CONS STR ABS) and STR are very precise, we find that the sufficient time is also `2 + 2*length(xs)`. Therefore we know that this is the *exact* time complexity.

5.4.3 Approximation and Safety

We need to show that the expression $\mathcal{T}_n[\![\mathrm{e}]\!]\alpha$ gives a lower-bound estimate to the cost of the lazy evaluation of e in context α. Once again the correctness argument relies on the appropriate operational interpretation of (safe) projection information. In the case of the sufficient-time analysis we begin with a safe upper-bound (the call-by-value version), and improve on this using the projection information. A similar informal argument can be applied for the necessary-time equations. In this case we begin with a safe lower-bound, namely *zero* cost, and we improve on this using theorem 5.4.1.

In a non-strict context the lower-bound must be taken to be zero since non-strict contexts *may or may not* require a value greater than $\perp$, and so an expression in such a context *may or may not* need to be evaluated. Proposition 5.4.4 below establishes this property. First we establish the following useful result:

PROPOSITION **5.4.3** *Given safe projection transformers $f^{\#1}, \ldots, f^{\#n}$*

$$\mathrm{ABS} \sqsubseteq \alpha \sqsubseteq \mathrm{ID} \;\Rightarrow\; \mathrm{ABS} \sqsubseteq (f^{\#i}\alpha) \sqsubseteq \mathrm{ID}$$

PROOF The projection transformers $f^{\#i}$ satisfy

$$\alpha(f(u_1, \ldots, u_n)) \sqsubseteq f(u_1, \ldots, \beta_i(u_i), \ldots, u_n)$$

for all functions f and objects $u_1 \ldots u_n$, where $\beta_i = (f^{\#i}\alpha)$.
Now suppose $\lightning \sqsubset u_1, \ldots, u_n$, then since f is naturally extended to domains containing $\lightning$, we must have $\lightning \sqsubset f(u_1, \ldots, u_n)$, and so

$$\mathrm{ABS} \sqsubseteq \alpha \sqsubseteq \mathrm{ID} \;\Rightarrow\; \lightning \sqsubset \alpha(f(u_1, \ldots, u_n))$$

Now suppose also that some $u_i = \perp$, then

$$\begin{aligned} \mathrm{ABS} \sqsubseteq \alpha \sqsubseteq \mathrm{ID} &\Rightarrow \lightning \sqsubset \alpha(f(u_1, \ldots, u_n)) \sqsubseteq f(u_1, \ldots, (\beta_i \perp), \ldots, u_n) \\ &\Rightarrow \lightning \sqsubset \beta_i \perp \\ &\Rightarrow \mathrm{ABS} \sqsubseteq (f^{\#i}\alpha) \sqsubseteq \mathrm{ID} \end{aligned}$$

□

PROPOSITION **5.4.4** *For every expression e,* $\mathrm{ABS} \sqsubseteq \alpha \sqsubseteq \mathrm{ID} \Rightarrow \mathcal{T}_n[\![e]\!]\alpha = 0$

PROOF Structural induction in e:

- *constants, identifiers* : Immediate from definition of $\mathcal{T}_n[\![\,]\!]$
- *conditional* : Immediate from definition of $\mathcal{T}_n[\![\,]\!]$ and $\hookrightarrow_n$
- *primitive functions* :

$$\begin{aligned} \mathcal{T}_n[\![p(e_1, \ldots e_n,)]\!]\alpha &= \mathcal{T}_n[\![e_1]\!](p^{\#1}\alpha) + \cdots + \mathcal{T}_n[\![e_n]\!](p^{\#n}\alpha) \\ &= 0 + \cdots + 0 \end{aligned}$$

(by prop. 5.4.3 and induction hypothesis)

- *non-primitive functions* :

$$\begin{aligned}
\mathcal{T}_s[\![f(e_1,\dots,e_n)]\!]\alpha &= \mathcal{T}_n[\![e_1]\!](f^{\#1}\alpha)+\cdots+\mathcal{T}_s[\![e_n]\!](f^{\#n}\alpha)\\
&\quad +cf(e_1,\dots,e_n,\alpha)\\
&= \mathcal{T}_n[\![e_1]\!](f^{\#1}\alpha)+\cdots+\mathcal{T}_s[\![e_n]\!](f^{\#n}\alpha)+0\\
&\quad (\text{by } \hookrightarrow_n)\\
&= 0+\cdots+0\\
&\quad (\text{by prop. 5.4.3 and induction hypothesis})
\end{aligned}$$

□

Safety

The correctness argument given for the necessary and sufficient-time analyses applies when the cost-expressions are well-defined. As we have seen, the sufficient-time expressions are not always *safe* in the sense that they are not always well-defined, even when the original expressions are. The main result of this section is that the necessary-cost programs enjoy better termination properties than the sufficient-cost programs. We state this property in the following way:

THEOREM **5.4.5** *Given mutually recursive functions $f_1,\dots f_m$, defined by equations:*

$$f_i(x_1,\dots,x_{n_i})=e_i$$

$i=1\dots m$, *then for all objects $u_1,\dots u_{n_i}$, and contexts α*

$$\alpha(f_i(u_1,\dots,u_{n_i}))\sqsupset\bot\Rightarrow cf_i(u_1,\dots,u_{n_i},\alpha)\sqsupset\bot$$

where cf_i is defined by the equation

$$cf_i(x_1,\dots,x_{n_i},\alpha)=\alpha\hookrightarrow_n 1+\mathcal{T}_n[\![e_i]\!]\alpha$$

PROOF We will assume, without loss of generality, that we have just a single recursive function f, defined by the equation

$$f(x_1,\dots,x_n)=e_f$$

The meaning of the pair of functions (f,cf) is the least fixed point of the pair

$$\begin{aligned}
(F\ g\ (x_1,\dots,x_n) &= e_f\{g/f\},\\
cF\ cg\ (x_1,\dots,x_n,\alpha) &= \alpha\hookrightarrow_n 1+\mathcal{T}_n[\![e_f]\!]\alpha\{g/f,cg/cf\})
\end{aligned}$$

To be more formal we could give a denotational semantics for the language by defining a function

$$E : exp \rightarrow env \rightarrow funenv \rightarrow \mathcal{D}$$

which gives meaning to expression *exp* in a given environment *env*, and function environment *funenv*. The meaning of the functions f and cf (given by the function environment *funenv*) is then the least fixed point of

$$\lambda\phi.(\lambda\rho.E[\![e_f]\!]\ \rho\ \phi, \lambda\rho.E[\![\alpha \hookrightarrow_n 1 + \mathcal{T}_n[\![e_f]\!]\alpha]\!]\ \rho\ \phi)$$

To be even more formal we could make a distinction between the *representations* of projections used in the cost-program, and the meta-interpretation that views them as actual projections.

However, this level of detail will not make the structure of our proof any clearer, so we use a rather more informal notation where the environments are represented using the expression-substitution notation, and the syntactic constructs are overloaded in the obvious way.

The proof proceeds by fixed point induction over the two recursive definitions simultaneously. Since we are working with lifted domains (containing $\lightning$), all functions (including the primitives) are strict in $\lightning$, so the first approximation to the pair (f, cf) must be the pair of functions

$$F^0(x_1, \ldots, x_n) = \begin{cases} \lightning & \text{if any } x_j = \lightning \\ \perp & \text{otherwise} \end{cases}$$

$$cF^0(x_1, \ldots, x_n, \alpha) = \begin{cases} \lightning & \text{if any } x_j = \lightning \\ \perp & \text{otherwise} \end{cases}$$

i.e. we are taking the least fixed point *above* the always-$\lightning$function. The i^{th} approximation is given by the pair

$$\begin{array}{lcl} (F^i(x_1, \ldots, x_n) & = & e_f\{F^{i-1}/f\}, \\ cF^i(x_1, \ldots, x_n, \alpha) & = & \alpha \hookrightarrow_n 1 + \mathcal{T}_n[\![e_f]\!]\alpha\{F^{i-1}/f, cF^{i-1}/cf\}) \end{array}$$

We prove the property $P(f, cf)$ by fixed point induction, where

$$P(g, cg) = \alpha(g(u_1, \ldots, u_n)) \sqsupset \perp \Rightarrow cg(u_1, \ldots, u_n, \alpha) \sqsupset \perp$$

for all objects $u_1, \ldots u_n$, and contexts α. Since we assume that α, g and cg are monotonic, P is *inclusive*— it holds in the limit.

Base case: $P(F^0, cF^0)$ holds trivially.

Inductive case: Prove $P(F^{i+1}, cF^{i+1})$ given $P(F^i, cF^i)$:

$$\begin{array}{lcl} F^{i+1}(u_1,\ldots,u_n) & \equiv & e_f\{F^i/f, u_1/x_1,\ldots,u_n/x_n\} \\ cF^{i+1}(u_1,\ldots,u_n,\alpha) & \equiv & \alpha \hookrightarrow_n 1 + \mathcal{T}_n[\![e_f]\!]\alpha\{F^i/f, cF^i/cf, u_1/x_1,\ldots,u_n/x_n\} \end{array}$$

unfolding function definitions. In the following we shall use σ to represent *both* substitutions $\{F^i/f, u_1/x_1,\ldots,u_n/x_n\}$ and $\{F^i/f, cF^i/cf, u_1/x_1,\ldots,u_n/x_n\}$ (since the second is just an extension of the first). We can thus restate $P(F^{i+1}, cF^{i+1})$ as

$$\text{For all } \alpha, u_1,\ldots,u_n \;\; \alpha((e_f)\sigma) \sqsupset \bot \Rightarrow (\alpha \hookrightarrow_n 1 + \mathcal{T}_n[\![e_f]\!]\alpha)\sigma \sqsupset \bot$$

By the definition of $\hookrightarrow_n$ it is sufficient to show

$$\text{For all } \alpha,\ \text{FAIL} \sqsubseteq \alpha \sqsubseteq \text{STR},\ u_1,\ldots,u_n \;\; \alpha((e_f)\sigma) \sqsupset \bot \Rightarrow (\mathcal{T}_n[\![e_f]\!]\alpha)\sigma \sqsupset \bot \qquad (5.1)$$

The inductive case proceeds by induction in the structure of e_f:

- $e_f \equiv$ *constants, identifiers* : RHS of 5.1 true by defn of $\mathcal{T}_n[\![]\!]$

- $e_f \equiv$ `if` e_1 `then` e_2 `else` e_3 : FAIL $\sqsubseteq \alpha \sqsubseteq$ STR gives us

$$\alpha((\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3)\sigma) \sqsupset \bot \Rightarrow (e_1)\sigma \sqsupset \bot \qquad (5.2)$$

$$\begin{array}{ll} & (\mathcal{T}_n[\![\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3]\!]\alpha)\sigma \\ = & \alpha \hookrightarrow_n (\mathcal{T}_n[\![e_1]\!]\text{STR})\sigma + (\texttt{if } e_1 \texttt{ then } \mathcal{T}_n[\![e_2]\!]\alpha \texttt{ else } \mathcal{T}_n[\![e_3]\!]\alpha)\sigma \\ = & (\mathcal{T}_n[\![e_1]\!]\text{STR})\sigma + (\texttt{if } e_1 \texttt{ then } \mathcal{T}_n[\![e_2]\!]\alpha \texttt{ else } \mathcal{T}_n[\![e_3]\!]\alpha)\sigma \qquad (5.3) \end{array}$$

Now

$$\begin{array}{lll} \alpha((\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3)\sigma) \sqsupset \bot & \Rightarrow & \text{STR}((e_1)\sigma) \sqsupset \bot \\ & \Rightarrow & (\mathcal{T}_n[\![e_1]\!]\text{STR})\sigma \sqsupset \bot \end{array}$$

by the structural I.H. Similarly

$$\begin{array}{l} \alpha((\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3)\sigma) \sqsupset \bot \Rightarrow \\ \quad (\texttt{if } e_1 \texttt{ then } \mathcal{T}_n[\![e_2]\!]\alpha \texttt{ else } \mathcal{T}_n[\![e_3]\!]\alpha)\sigma \sqsupset \bot \end{array}$$

by cases according to $(e_1)\sigma$ and the structural I.H.

- $e_f \equiv p(e_1,\ldots,e_n)$:

$$(\mathcal{T}_n[\![p(e_1,\ldots,e_n)]\!]\alpha)\sigma = (\mathcal{T}_n[\![e_1]\!](p^{\#1}\alpha) + \cdots + \mathcal{T}_n[\![e_n]\!](p^{\#n}\alpha))\sigma$$

Split into cases according to $(p^{\#i}\alpha)$, $i = 1\ldots n$

(i) ABS $\sqsubseteq (p^{\#^i}\alpha) \sqsubseteq$ ID: Then $\mathcal{T}_n[\![e_i]\!](p^{\#^i}\alpha) = 0 \sqsupset \bot$ by prop. 5.4.4

(ii) FAIL $\sqsubseteq (p^{\#^i}\alpha) \sqsubseteq$ STR:

$$\begin{array}{rll}
 & \alpha(p(e_1,\ldots,e_n))\sigma \sqsupset \bot & \\
\Rightarrow & p(e_1,\ldots,(p^{\#^i}\alpha)e_i,\ldots,e_n)\sigma \sqsupset \bot & \text{(safety of } p^{\#^i}) \\
\Rightarrow & ((p^{\#^i}\alpha)e_i)\sigma \sqsupset \lightning & (p \text{ strict in } \lightning) \\
\Rightarrow & ((p^{\#^i}\alpha)e_i)\sigma \sqsupset \bot & \text{(lift-strict projection } (p^{\#^i}\alpha)) \\
\Rightarrow & (\mathcal{T}_n[\![e_i]\!](p^{\#^i}\alpha))\sigma \sqsupset \bot & \text{(structural I.H.)}
\end{array}$$

So together,

$$\begin{array}{rcl}
\alpha(p(e_1,\ldots,e_n))\sigma) \sqsupset \bot & \Rightarrow & (\mathcal{T}_n[\![e_i]\!](p^{\#^i}\alpha))\sigma \sqsupset \bot \\
 & \Rightarrow & (\mathcal{T}_n[\![p(e_1,\ldots,e_n)]\!]\alpha)\sigma \sqsupset \bot
\end{array}$$

as required.

- $e_f \equiv f(e_1,\ldots,e_n)$:

$$\begin{array}{l}
(\mathcal{T}_n[\![f(e_1,\ldots,e_n)]\!]\alpha)\sigma \\
\quad = \quad (cf(e_1\ldots e_n,\alpha) + \mathcal{T}_n[\![e_1]\!](f^{\#^1}\alpha) + \cdots + \mathcal{T}_n[\![e_n]\!](f^{\#^n}\alpha))\sigma \\
\quad = \quad (cf(e_1\ldots e_n,\alpha))\sigma + (\mathcal{T}_n[\![e_1]\!](f^{\#^1}\alpha) + \cdots + \mathcal{T}_n[\![e_n]\!](f^{\#^n}\alpha))\sigma
\end{array}$$

As in the case of the primitive functions we have

$$\begin{array}{rl}
 & \alpha(f(e_1,\ldots,e_n)\sigma) \sqsupset \bot \\
\Rightarrow & (\mathcal{T}_n[\![e_i]\!](f^{\#^i}\alpha))\sigma \sqsupset \bot \\
\Rightarrow & (\mathcal{T}_n[\![e_1]\!](f^{\#^1}\alpha) + \cdots + \mathcal{T}_n[\![e_n]\!](f^{\#^n}\alpha))\sigma \sqsupset \bot
\end{array} \tag{5.4}$$

Now, $(cf(e_1\ldots e_n,\alpha))\sigma \equiv cF^i(e_1\sigma,\ldots,e_n\sigma,\alpha))$, substituting using σ. And so by the main (fixed-point) inductive hypothesis, $P(F^i, cF^i)$, we have, whenever α is lift-strict:

$$\alpha(f(e_1,\ldots,e_n)\sigma) \sqsupset \bot \Rightarrow (cf(e_1\ldots e_n,\alpha))\sigma \sqsupset \bot$$

Together with 5.4 gives us

$$\alpha(f(e_1,\ldots,e_n)\sigma) \sqsupset \bot \Rightarrow (\mathcal{T}_n[\![f(e_1,\ldots,e_n)]\!]\alpha)\sigma \sqsupset \bot$$

as required

The proof extends in a straightforward way to an arbitrary set of mutually recursive functions by performing fixed-point induction over all the functions simultaneously.

□

5.5 Higher-Order Lazy Time Analysis

In this section we develop an extension to the techniques for lazy time analysis, to incorporate higher-order functions. This is achieved by adaptation of the higher-order analysis given in chapter 3, together with a conservative extension to the context information available for first-order functions.

5.5.1 Context Information

The extension of lazy-time analysis to higher-order functions necessitates the use of projection information. Here we immediately run into some problems: The techniques which we have assumed so far, concerning the form and derivation of projection transformers, are not directly extendible to higher-order functions. If we consider, for example, an instance of the apply function, `apply f x`, in some context α. The problem here is that there is no useful context information that can be propagated to x (by any context function $\texttt{apply}^{\#2}$) which is independent of the function f.

For the purposes of this chapter it will not be necessary to introduce these devices. We will demonstrate a method for time analysis that works with a very simple, but not very concise, extension of the context information to higher-order functions. The extension to projection information required gives very poor context-information in the presence of higher-order functions, but we counteract this deficiency by using a variant of the cost-closure technique developed in chapter 3.

5.5.2 Language

The language we use here is defined by the same grammar as that of the higher-order language of chapter 3:

$$\begin{aligned} exp &::= exp\; e \mid e \\ e &::= \texttt{if}\; e_1\; \texttt{then}\; e_2\; \texttt{else}\; e_3 \mid (exp) \mid f_i \mid \\ &\quad\;\; p_i \mid x \mid c \end{aligned}$$

Where we have function definitions of the form

$$f_i\, e_1 \ldots e_{n_i} = exp_i$$

along with curried primitive functions p_i (of arity m_i).

5.5.3 The Projection Transformers

The method we shall describe for constructing the time equations will require the use of the same style of projection transformers that are used for the first-order analysis: for each function definition f_i we will require projection transformers ${f_i}^{\#k}$ such that

$$\alpha(f_i\, u_1 \ldots u_n) \sqsubseteq f_i\, u_1 \ldots (\beta\, u_k) u_{k+1} \ldots u_{n_i}$$

for all objects u, where $\beta = ({f_i}^{\#k} \alpha)$.

Since we are working with a higher-order language, we may have expressions of the form

$$f_i\, e_1 \ldots e_{n_i} e_{n_i+1} \ldots e_m$$

Here the contexts propagated to expressions $e_1 \ldots e_{n_i}$ are determined by the projection transformers of f_i. For a conservative estimate we know it is safe to propagate the context ID to the expressions $e_{n_i+1} \ldots e_m$. In fact, we will show how the analysis we present will be able to use more concise information in this instance.

Objects of function type will also require projections to describe the context in which they are needed. A projection of a function gives a function which has less defined results on some of its arguments. For the purpose of time analysis we will use the four-point context domain: In an expression of the form *exp e* in a context α, we can safely set the context for *exp* to be a mapping of α into the four-point domain for functions such that ABS and FAIL are preserved, strict contexts are mapped onto STR, and non-strict onto ID. For convenience we define a functional $\Diamond$ to perform this task:

Definition **5.5.1**

$$\Diamond\alpha = \begin{cases} \text{FAIL} & \textit{if } \alpha = \text{FAIL} \\ \text{ABS} & \textit{if } \alpha = \text{ABS} \\ \text{STR} & \textit{if } \text{FAIL} \sqsubset \alpha \sqsubseteq \text{STR} \\ \text{ID} & \textit{if } \text{ABS} \sqsubset \alpha \sqsubseteq \text{ID} \end{cases}$$

□

5.5.4 Accumulating Cost-Functions

We will only present a sufficient time analysis. The dual necessary-time analysis is easily definable, and has the appropriate termination properties, but for technical reasons discussed later does not always give a true lower-bound on full lazy evaluation time.

As in the strict higher-order language we will define, for each function in the language, a cost function which is constructed via two syntactic maps. The first, $\mathcal{V}_L[\![\cdot]\!]$, plays the same rôle as that of $\mathcal{V}$ in the higher-order strict language — it constructs cost-closures and makes their application explicit via an apply function $\mathtt{@}_L$. The second, $\mathcal{T}_L[\![\cdot]\!]$, is used to define the cost-expressions, which in this case are functions from context to cost. In the following we will use the term *cost-expression* to refer to objects of type *context* $\rightarrow$ *cost*.

User-defined functions

For each function defined $f_i\, x_1 \ldots x_{n_i} = e_i$ we define a sufficient-cost function to be

$$\begin{aligned} cf_i\,\langle x_1, c_1\rangle \ldots \langle x_{n_i}, c_{n_i}\rangle\, \alpha = \alpha \hookrightarrow_s 1 \;&+\; \mathcal{T}_L \circ \mathcal{V}_L[\![e_i]\!]\,\alpha \\ &+\; c_1({f_i}^{\#1}\alpha) \\ &+\; \ldots \\ &+\; c_{n_i}({f_i}^{\#n_i}\alpha) \end{aligned}$$

In addition to the context-transformers, the cost-functions refer to modified versions of the functions themselves:

$$f_i'\, x_1 \ldots x_{n_i} = \mathcal{V}_L[\![e_i]\!]$$

The definitions of $\mathcal{V}_L$ and $\mathcal{T}_L$ are given in figures 5.5 and 5.6. These definitions will be explained in the following sections:

Application and its Cost

The cost-functions defined above now have additional parameterisation in the form of cost-expressions paired with each argument. We will explain this choice by considering the cost associated with function application *exp e*.

In the higher-order strict language, application is first translated to $exp' \mathbin{\mathtt{@}} e'$ (where exp' is defined according to $\mathcal{V}$) and the cost of evaluation is

$$\mathcal{T}[\![exp']\!] + \mathcal{T}[\![e']\!] + exp' \mathbin{\mathtt{c@}} e'$$

Suppose we begin by re-using $\mathcal{V}$, and we attempt to define (with respect to some context β) a lazy version of $\mathcal{T}$, $\mathcal{T}_L$.

In the rule for $\mathcal{T}_L[\![exp' \mathbin{\mathtt{@}} e']\!]\,\beta$ we must propagate the context β to the appropriate cost-expressions. As we noted in section 5.5.3, we can map β into a four-point domain $\Diamond\beta$ to get a safe context for exp'. We do not know the appropriate context for e', but we can always safely use the context ID and set

$$\mathcal{T}_L[\![exp' \; @ \; e']\!]\beta = \mathcal{T}_L[\![exp']\!]\Diamond\beta + \mathcal{T}_L[\![e']\!]\text{ID} + (exp' \; \texttt{c@} \; e')\,\beta$$

Two major problems make such a rule unsatisfactory.

(i) No useful context information is propagated to e'. The information we have available is the projection transformers, but this is not used since we do not in general know which projection transformer is appropriate.

(ii) If we have a partial application, for example if exp is `cons` (and so exp' is `(cons,ccons,2)`) then e may not be evaluated at all.

We solve both of these problems by passing both the argument, *and* the cost-expression to the cost-function. It is then the cost-function's task to apply the appropriate context (which is determined by the projection transformers of the function) to these cost-expressions—see the cost-function scheme above. We introduce new versions of `@` and `c@` to accommodate these requirements.

Cost-closures and the apply function

For these reasons we need to define a new version of $\mathcal{V}$ and a different version of the function `@`. The version of $\mathcal{V}$, $\mathcal{V}_L$ is defined in figure 5.5. Because, in the rule for application, the cost-closure $\mathcal{V}_L[\![exp]\!]$ is applied to an expression cost-expression pair $\langle e\,,ce\rangle$, we need a new version of the `@` function which satisfies:

$$(f, cf, n) \; @_L \; \langle e\,,ce\rangle = \begin{cases} f \; e & \text{if } n = 1 \\ (f \; e, cf \; \langle e\,,ce\rangle, n-1) & \text{otherwise} \end{cases}$$

Note that cost-closures retain the same *function–costfunction–arity* structure.

Defining the cost-expressions

Figure 5.6 also defines cost-expressions via a mapping $\mathcal{T}_L[\![\,\cdot\,]\!]$. A significant difference here is that we do not make the definition with respect to a particular context. This is because we wish to pass cost-expressions (functions *context* $\rightarrow$ *cost*) to the cost-functions without binding them to a particular context.

To define $\mathcal{T}_L$we introduce some useful functional "combining forms" for cost-expressions:

- Addition of cost expressions: in order to add cost expressions we need to define an appropriate functional $\oplus$ such that

  $$(ce_1 \oplus ce_2)\,\alpha = (ce_1\,\alpha) + (ce_2\,\alpha)$$

  were ce_1 and ce_2 are cost-expressions, and α is a context.

  In fact in the definition of $\mathcal{T}_L$we use a more specialised addition operator, $\diamond\!\!+$, which (for the left operand) maps the context into the four-point projection domain of the left operand. We can safely map any projection over $\mathcal{D}$, α into the four-point sub-lattice of $\mathcal{D}$. $\diamond\!\!+$ can be defined by the equation

  $$(ce_1 \diamond\!\!+ ce_2)\,\alpha = (ce_1(\diamond\alpha)) + (ce_2\,\alpha)$$

  By allowing the terms ID, STR, ABS and FAIL to be "polymorphic", then $\diamond\!\!+$ is associative.

- Constant functions: We need some constant cost-expressions which give the same cost in any non-ABS context: For each non-negative integer k we have the cost-expression $\overline{k}$ such that

  $$\overline{k}\;\alpha = \alpha \hookrightarrow_s k$$

  In the rules for $\mathcal{T}_L$we only use the function $\overline{0}$, which satisfies $\overline{0}\;\alpha = 0$ for any context α.

Consider the rule for application:

$$\mathcal{T}_L[\![exp' \;@_L\; \langle e', ce'\rangle]\!] = \mathcal{T}_L[\![exp']\!] \diamond\!\!+ (exp' \;\texttt{c@}_L\; \langle e', ce'\rangle)$$

If we apply this expression to a context β, we get

$$\mathcal{T}_L[\![exp' \;@_L\; \langle e', ce'\rangle]\!]\beta = \mathcal{T}_L[\![exp']\!]\;\diamond\beta + (exp' \;\texttt{c@}_L\; \langle e', ce'\rangle)\;\beta$$

To ensure $\texttt{c@}_L$ gives us a cost expression, only a small change is needed from the definition of $\texttt{c@}$

$$(f, cf, n)\;\texttt{c@}_L\;\langle e, ce\rangle = \begin{cases} cf\;\langle e, ce\rangle & \text{if } n = 1 \\ \overline{0} & \text{otherwise} \end{cases}$$

Primitive functions

The cost-function associated with a primitive function p_i of arity m_i is

$$cp_i\;\langle x_1, c_1\rangle \ldots \langle x_{m_i}, c_{m_i}\rangle\;\alpha = c_1({p_i}^{\#1}\alpha) + \ldots + c_{m_i}({p_i}^{\#m_i}\alpha)$$

$$\begin{array}{rcl}
\mathcal{V}_L[\![exp\ e]\!] & = & \mathcal{V}_L[\![exp]\!]\ \texttt{@}_L\ \langle \mathcal{V}_L[\![e]\!], \mathcal{T}_L \circ \mathcal{V}_L[\![e]\!] \rangle \\
\mathcal{V}_L[\![\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3]\!] & = & \texttt{if}\ \mathcal{V}_L[\![e_1]\!]\ \texttt{then}\ \mathcal{V}_L[\![e_2]\!]\ \texttt{else}\ \mathcal{V}_L[\![e_3]\!] \\
\mathcal{V}_L[\![(exp)]\!] & = & (\mathcal{V}_L[\![exp]\!]) \\
\mathcal{V}_L[\![f_i]\!] & = & (f'_i\ ,\ cf_i\ ,\ n_i\ ) \\
\mathcal{V}_L[\![p_i]\!] & = & (p_i\ ,\ cp_i\ ,\ m_i\ ) \\
\mathcal{V}_L[\![c]\!] & = & c \\
\mathcal{V}_L[\![x]\!] & = & x
\end{array}$$

Figure 5.5: The function modification map $\mathcal{V}_L$

$$\begin{array}{rcl}
\mathcal{T}_L[\![exp'\ \texttt{@}_L\ \langle e', ce' \rangle]\!] & = & \mathcal{T}_L[\![exp']\!] \diamond\!\!+ (exp'\ \texttt{c@}_L\ \langle e', ce' \rangle) \\
\mathcal{T}_L[\![\texttt{if}\ e'_1\ \texttt{then}\ e'_2\ \texttt{else}\ e'_3]\!] & = & \mathcal{T}_L[\![e'_1]\!] \diamond\!\!+\ \texttt{if}\ e'_1\ \texttt{then}\ \mathcal{T}_L[\![e'_2]\!]\texttt{else}\ \mathcal{T}_L[\![e'_3]\!] \\
\mathcal{T}_L[\![(exp')]\!] & = & (\mathcal{T}_L[\![exp']\!]) \\
\mathcal{T}_L[\![(p_i\ ,\ cp_i\ ,\ m_i\ )]\!] & = & \overline{0} \\
\mathcal{T}_L[\![(f_i\ ,\ cf_i\ ,\ n_i\ )]\!] & = & \overline{0} \\
\mathcal{T}_L[\![c]\!] & = & \overline{0} \\
\mathcal{T}_L[\![x]\!] & = & \overline{0}
\end{array}$$

Figure 5.6: The cost-function construction map $\mathcal{T}_L$

5.5.5 Examples

The cost-functions constructed are most easily explained by example. Consider a function called `composeall` which, given a list of functions gives the composition of those functions

```
composeall fs x  =  if (null fs) then x
                    else hd fs (composeall (tl fs) x)
```

After removing the trivial instances of `c@`$_L$ and `@`$_L$, and the "zero-costs", the derived cost-function is

```
ccomposeall <fs,fsc> <x,xc> α =
   α ↪s1 + (if (null fs) then 0
            else hd fs c@L < composeall' (tl fs) x ,
                             ccomposeall <(tl fs),0> <x,0>
                             > )
           + fsc (composeall#1 α)
           + xc (composeall#2 α)
```

Where

```
composeall' fs x  =  if (null fs) then 0
                     else hd fs @L <
                               composeall' (tl fs) x ,
                               ccomposeall <(tl fs),0̄> <x,0̄>
                               >
```

Consider `composeall` in a strict context. It is straightforward to verify that

$$\texttt{composeall}^{\#1}\,\mathrm{STR} = \mathrm{NIL} \sqcup \mathrm{CONS\ STR\ ID}$$

This says that in a strict context, the list `fs` is required to be either `nil`, or a `cons` whose head will be needed in a strict context—alternatively we can say `composeall` is *head-strict*.

Also we can (as for any projection transformer) set

$$\texttt{composeall}^{\#2}\,\alpha = \begin{cases} \mathrm{ABS} & \text{if } \alpha = \mathrm{ABS} \\ \mathrm{ID} & \text{otherwise} \end{cases}$$

These projection transformers will be sufficient for use in the following example. `composeall`$^{\#1}$ and `composeall`$^{\#2}$ are abbreviated to #1 and #2 respectively.

Now we wish to evaluate the cost of the expression

```
composeall [(K 9), fact, fact ] m
```

in the context STR, where we use the usual shorthand for list expressions, `fact` is the ubiquitous factorial function (whose definition we omit), and m is some arbitrary integer value.

`K` is a K-combinator of type: *integer* $\rightarrow$ *integer* $\rightarrow$ *integer*

```
K x y  =  x
```

Since `K` ignores its second argument and returns its first, it is easy to see that it is safe to use the following projection transformers:

$$\begin{aligned} \mathtt{K}^{\#1}\ \alpha &= \alpha \\ \mathtt{K}^{\#2}\ \alpha &= \mathrm{ABS} \end{aligned}$$

The derived cost-function is

$$\begin{aligned} \mathtt{cK}\ \mathtt{<x,xc>}\ \mathtt{<y,yc>}\ \alpha &= \alpha \hookrightarrow_s 1\ \mathtt{+}\ \mathtt{xc}\ (\mathtt{K}^{\#1}\ \alpha) \\ &\qquad\qquad \mathtt{+}\ \mathtt{yc}\ (\mathtt{K}^{\#2}\ \alpha) \\ &= \alpha \hookrightarrow_s 1\ \mathtt{+}\ \mathtt{xc}\ \alpha \end{aligned}$$

The cost of evaluating the above expression is thus (using $\mathtt{fact}^{cc}$ to denote the cost-closure `(fact',cfact,1)`)

$\mathcal{T}_L \circ \mathcal{V}_L[\![$`composeall [(K 9),fact,fact]` m $]\!]$ STR

$=$ `(ccomposeall <[(K 9,cK <9,`$\overline{0}$`>,1),fact`^{cc}`,fact`^{cc}`] ,`$\overline{0}$` >`
`<`m`,`$\overline{0}$`> )` STR

$=$ `1 + (K 9,cK <9,`$\overline{0}$`>,1)c@`$_L$
`<composeall' [fact`^{cc}`,fact`^{cc}`]` m`,`
`ccomposeall <[fact`^{cc}`,fact`^{cc}`],`$\overline{0}$`> <`m`,`$\overline{0}$`>`
`>` STR
`+` $\overline{0}$`(`$\#1$ STR`) +` $\overline{0}$`(`$\#2$ STR`)`

$=$ `1 + cK <9,`$\overline{0}$`> <composeall' [fact`^{cc}`,fact`^{cc}`]` m`,`
`ccomposeall <[fact`^{cc}`,fact`^{cc}`],`$\overline{0}$`> <`m`,`$\overline{0}$`>`
`>` STR

$=$ `1 + 1 +` $\overline{0}$ STR

$=$ `2`

□

5.5.6 Correctness

It is possible to give a correctness argument based on the relation between lazy evaluation and call-by-value evaluation orders, together with the correctness results of chapter 3, much as we did for the first-order sufficient-time analysis. However, to do this properly we would need to give a slightly different version of the call-by-value analysis using the accumulating cost-function approach, where cost-functions are parameterised on argument-cost pairs. This version would then need to be shown correct with respect to a slightly lazier version of the semantics in which a partial application incurs *no* evaluation. Such modifications are quite straightforward, but are somewhat lengthy and are not pursued here.

5.5.7 Limitations and Improvements

The use of first-order context analysis in the analysis of a higher-order language means that, even though cost-expressions are passed as arguments (so they can be applied to the appropriate context), there are many cases where the contexts derived for higher-order functions are not sufficiently concise. Consider the following function definitions:

```
apply f x  =  f x
ignor x    =  7
```

For satisfiable contexts α we have the following projection transformers:

$$\begin{aligned}\texttt{apply}^{\#1}\alpha &= \Diamond\alpha\\ \texttt{apply}^{\#2}\alpha &= \textsc{id}\\ \texttt{ignor}^{\#1}\alpha &= \textsc{abs}\end{aligned}$$

Without knowing about the context of the function `apply`, the context for x is approximated by the least informative context ID.

The sufficient-time equations constructed with these projection transformers are

$$\begin{aligned}\texttt{capply <f,fc> <x,xc>}\ \alpha &= \alpha \hookrightarrow_s \texttt{1 + fc(}\Diamond\alpha\texttt{) + xc ID}\\ &\qquad \texttt{+ (f c@ <x,}\overline{0}\texttt{>)}\ \alpha\\ \\ \texttt{cignor <x,xc>}\ \alpha &= \alpha \hookrightarrow_s \texttt{1 + xc (ignor}^{\#1}\alpha\texttt{)}\\ &= \alpha \hookrightarrow_s \texttt{1 + xc ABS}\\ &= \alpha \hookrightarrow_s \texttt{1}\end{aligned}$$

Now consider the expression

```
apply ignor expensive
```

evaluated in the context STR (where "`expensive`" is some arbitrary, potentially costly expression):

$$\begin{aligned}&\mathcal{T}_L \circ \mathcal{V}_L[\![\texttt{apply ignor expensive}]\!]\,\textsc{str}\\ &= \texttt{capply <(ignor',cignor,1),}\overline{0}\texttt{> <}e\texttt{,}ce\texttt{> STR}\\ &= \texttt{1 + (}\overline{0}\texttt{ STR) + (}ce\texttt{ ID) + ((ignor',cignor,1) c@ <}e\texttt{,}\overline{0}\texttt{>)STR}\\ &= \texttt{1 + (}ce\texttt{ ID) + cignor <}e\texttt{,}\overline{0}\texttt{> STR}\\ &= \texttt{2 + (}ce\texttt{ ID)}\end{aligned}$$

where $\texttt{<}e\texttt{,}ce\texttt{>} = \texttt{<}\mathcal{V}_L[\![\texttt{expensive}]\!]\texttt{,}\mathcal{T}_L \circ \mathcal{V}_L[\![\texttt{expensive}]\!]\texttt{>}$

A possibility here is that we could parameterise the cost-function further by including the projection-transformer for `f`. This is essentially the "second-order" strictness analysis of [Wra86], and does not extend to a fully higher-order analysis in an obvious way.

Propagating Cost-Expressions

As we can see, the lack of accurate projection transformers means that the cost-expression ce is applied to the imprecise context ID. However, we can rectify this problem by utilising the technique of passing cost-expressions so that they reach their context. The expression bound to `x` in the function `apply` is evaluated in the context of the function bound to `f`, so we can pass the cost-expression on to the cost-function associated with `f` as follows:

$$\texttt{capply <f,fc> <x,xc>}\ \alpha \;\texttt{=}\; \alpha \hookrightarrow_s 1\ \texttt{+ fc(}\Diamond\alpha\texttt{)}$$
$$\texttt{+ (f c@ <x,xc>)}\ \alpha$$

Now if we use this version of `apply` in the above example we get:

$$\begin{aligned}
&\mathcal{T}_L \circ \mathcal{V}_L[\![\texttt{apply ignor expensive}]\!]\,\text{STR}\\
&= \texttt{capply <(ignor',cignor,1),}\overline{0}\texttt{>}\ \texttt{<}e\texttt{,}ce\texttt{>}\ \text{STR}\\
&= \texttt{1 + (}\overline{0}\ \text{STR}\texttt{) + ((ignor',cignor,1) c@ <}e\texttt{,}ce\texttt{>)}\text{STR}\\
&= \texttt{1 + cignor <}e\texttt{,}ce\texttt{>}\ \text{STR}\\
&= \texttt{2}
\end{aligned}$$

which is exactly equal to the necessary-cost as expected.

To generalise this technique we must check that any parameter whose cost-expression we wish to propagate is not shared (*i.e.* it is not required in more than one context). For a sufficient-time analysis we could propagate to all contexts, while in a necessary-time analysis we could chose to propagate the cost-expression to a single context. In addition we need to determine when the propagation is necessary, since unnecessary propagation (*i.e.* when the context information is sufficiently concise) decreases the compositionality of cost-functions.

5.6 Conclusions

We have presented a method of analysing the time complexity of a lazy higher-order functional language. The first-order analysis is based upon [Wad88]: projections are used to characterise the context in which an expression is evaluated, and cost-equations are parameterised by this context-description. We have introduced two types of time-equation: *sufficient-time* equations (corresponding to the equations in [Wad88]), and *necessary-time* equations, which together provide bounds on the exact time-complexity. In the rest of this chapter we consider a closely related approach for the analysis of a first-order language, and discuss some of the remaining problems in the area.

5.6.1 Related Work

As we discussed in the introduction, Bjerner's time analysis for programs in the language of Martin-Löf type-theory [Bje89] is relevant to the analysis of first-order lazy functional languages, and provided inspiration for Wadler's work. His operational model of contexts, *evaluation degrees* are operational in nature, and give a more direct handle on the correctness issues of the time analysis. It is not clear that the method (from the viewpoint of the correctness issues) extends easily to a general recursive language, and so it is notable that more recently, Bjerner and Holmström [BH89] have adapted the overall ideas in [Bje89] to give a calculus for the time analysis of a first-order functional language which is, like the projection approach, based on a more denotational notion of context. This description is called a *demand*, and it is a *representation* of an approximation to a value. Given a demand δ, which represents an approximation to the result of an expression required, a *demand analysis* is presented which gives the demands corresponding to *the least* approximation to the (value) environment necessary to obtain at least δ's-worth of the expression. The demand analysis is then used in the definition of a compositional time analysis much in the same way that projections are used in [Wad88] and the sufficient-time analysis given here. There is a demand ϕ which is a representation of $\perp$. If ϕ is a correct demand for some sub-expression, then it is safe to replace the value of the expression with the approximation $\perp$, and so the cost due to this expression is zero. In this way we see that ϕ plays the rôle of the projection ABS.

Relating demands to projections, we can see that projections are more general: a demand directly represents a safe approximation to a value, whereas a projection when applied to a value, returns a safe approximation. For every demand δ (in the language of demands defined in [BH89]) which represents some value d, there is a corresponding projection (ignoring typing issues) Δ which sends every value greater than or equal to d to d, and sends any other value to $\lightning$. Given this correspondence, the language of demands induces a language of projections. If we have some projection transformers that operate on projections in this language then it is straightforward to show that sufficient and necessary-time equations are equivalent, hence verifying that the method in [BH89] specifies exact time-cost. With this observation, relating projections and demands, we also have a very close correspondence between the main correctness result developed (independently) in [BH89] (apart from the correctness of the demand analysis), and the necessary-time safety result of theorem 5.4.5.

The lack of generality of demands has both advantages and disadvantages. In their favour, the simple language of demands leads to a description of demand analysis which not only describes the best (smallest) safe demands, but is also an algorithm for computing them. On the other hand demands are in some ways too precise: to analyse a program one must find a demand which corresponds to (an approximation) to the value program. Together with a lack of a notion of approximation, this sometimes makes demands difficult to reason about.

An advantage of projections is that they allow us to express "amounts of evaluation" in a much more general way, such as the projection STR which describes *some* value greater than bottom. With these more general descriptions come other problems, which are considered below.

5.6.2 Determining Appropriate Projections

The method we have outlined relies on projection information which for convenience we have assumed comes in the shape of projection transformers. An issue which we have not considered in any detail is how to determine this safe projection information. One appeal of the projection based approach is that we know we are able to obtain *some* safe projections by the application of the automatic techniques from [WH87]. This is appealing, but the degree of approximation in this approach can lead to poor time analyses. It is not difficult, however, to give a more exact version of the projection transformer equations from [WH87] (where the standard semantics is appealed to in the case of the conditional expression). This approach is the basis of the "extended projection transformers" suggested in [Wad88]. This approach gives a method for reasoning about more precise projection equations, but unlike the demand approach cited above, does not constitute an *algorithm* for computing the safe projections *i.e.* it is a mathematical semantics. Throughout this thesis we have emphasised the algorithmic flavour of our calculi for time analysis, via the expression of program properties using the language itself, and it would be nice if the projection information could be expressed in a similar algorithmic way. It is clear that to achieve this we need to work (more explicitly) with *representations* of projections. As we noted above, the language of demands from [BH89] can be viewed as a projection language, but not a very expressive one. Some generalisations of this language are straightforward, such as the addition of a term corresponding to STR[4]. Ideally

[4]Interestingly, this addition gives essentially the language of *patterns* from [JM89a] which are used in a compile-time garbage-collection, and in a form of backwards strictness analysis [JM89b]. In fact, the *necessity patterns* from [JM89a] together with the *strictness patterns* from [JM89b] could also form the basis for necessary and sufficient time analyses respectively.

we would like a language that contains some sort of recursion operator to allow us to describe useful projections over recursive data-structures. The language of *rational strictness patterns* from [HW89] contains relevant ideas, where the rational patterns are acyclic graph structures. Some similar issues are considered in [Lau89] where projections are used to describe binding-times in a partial evaluator. To permit a computable binding-time analysis the language used is not particularly general, but some of the issues raised in the implementation of the analysis (such as uniqueness of representation) suggest that generalisations to the language of projections are incompatible with the "algorithmic" aim.

5.6.3 Higher-Order Analysis

The approach to higher-order functions has brought together many strands of our research programme. Improvements to the methods we have proposed are possible, and some of these have been considered in section 5.5.7. The overall approach has been to apply the denotational notions of projections for the first-order parts of the program, and use the more operational notion of (cost) closures to deal with the higher-order parts. This approach is in the spirit of some other program-analysis methods, for example [Mog88, Jen90], which, within a denotational framework, deal with the higher-order functions in an operational manner via abstract representations of closures.

It is expected that better descriptions of context in the presence of higher-order functions will give significant improvements to the quality of the time-equations. We can look to the active area of strictness analysis (from which the projection ideas come) for possibilities. Wray's thesis [Wra86] shows how to handle a "second order" language (for strictness analysis) by additional parameterization of the context transformers to include the context transformers for functional arguments. An approach to fully higher-order backwards analysis is outlined in [Hug87]. This is based on a complex mixture of abstract interpretation (forwards analysis) and first-order backwards analysis, but the details have yet to be worked out. The most promising approach is Hunt's generalisation of projections to *partial equivalence relations* (PERs) with which a higher-order abstract-interpretation is defined [Hun90a] via the logical-relations framework of [Abr90a].

Chapter 6

Conclusions

In this chapter we conclude by summarising the contributions of this thesis. Directions for future work, both direct extensions to this work, and application areas are considered.

6.1 Summary

In this thesis we have addressed issues that arise in the time-analysis of functional programs. The areas of attention have been those (essential) features of functional languages for which traditional methods for analysing complexity are inadequate: specifically, higher-order functions and lazy evaluation.

6.1.1 Higher-order functions

The analysis of higher-order functions falls outside the scope of traditional methods because an expression in a higher-order language has potentially many facets to it's complexity: the complexity of it's evaluation, the complexity of subsequent applications, applications of applications, and so forth.

In chapter 3 we developed a calculus for reasoning about cost in a higher-order functional language with a call-by-value calling mechanism. Translation schemes were developed for the mechanical construction of *cost-functions* from the functions in a program. The construction of cost-functions is *uniform*, in the sense that it is not dependent on the ways in which function definitions are *used* in a program. The expression of cost-functions in the language under analysis has several initial advantages:

- Techniques and tools for reasoning about and manipulating functional programs are applicable (as illustrated in some of the work on the mechanisation of the analysis of first-order languages [LeM88b, Ros89]).

- Informal reasoning and propositions about cost can be tested against the execution of cost-functions, and (partially optimised) cost-functions can provide implementation independent profiling.

Illustrating the first point, a process called *factorisation* was introduced. Factorisation is a simple tactic for reasoning about the cost of higher-order functions with limited knowledge of the functional parameters with which they are called.

The key to the development of cost-functions was the introduction of *cost-closures*. Cost-closures are syntactic structures which enable intensional information to be carried around with function-valued objects. The maintenance of cost-closures is via *arity* considerations, and is (therefore) applicable to untypable functions.

6.1.2 Lazy evaluation

Three complementary calculi for reasoning about non-strict evaluation have been developed, which are summarised and contrasted below.

Analysis via translation At the end of chapter 3 we showed how the methods developed for the analysis of a higher-order call-by-value language could be applied to the analysis of a call-by-name language. The key to this method is the use of a cost-preserving translation from the call-by-name language to call-by-value. Higher-order functions can be used to provide relatively straightforward translations based on well-known implementation techniques. An advantage of this approach is that the techniques of chapter 3 can be applied. A disadvantage is that the translated programs can be unwieldy, making them unsuitable for reasoning by hand.

An operational calculus for time analysis In chapter 4 a simple operational semantics for a call-by-name language with lazy lists was used as a basis for a calculus for time analysis. The main difference from the preceding approaches is that time-equations are developed directly from the operational model, instead of an indirect derivation of cost-functions. The advantage of this approach is that the time-equations are sufficiently concise to reason about cost by hand; the initial disadvantage is that ordinary equational reasoning on expressions is not valid within this calculus. A theory of cost-simulation is developed to give a suitable notion of equivalence between expressions so that the time equations can be extended with additional equational laws, cost-equivalences, which obey the "Leibniz principal" of substitution within the time equations. The main attraction of this approach is its simplicity. The time equations together with some cost-equivalences form the easiest route to reasoning about small programs.

Lazy time analysis The disadvantages of the above approach for analysing the time-cost of lazy programs are:

(i) The direct operational route means that the calculus models call-by-name more easily than call-by-need.

(ii) The methods are not compositional. In the case of the translation method, the analysis of the cost of evaluating a sub-program will tell us the cost of building the closure corresponding to that object. Interesting cost properties can only be determined by placing the expression in some context. Similarly, the operational calculus of chapter 4 is for reasoning about *programs* and does not permit formal reasoning about sub-expressions out of context.

Chapter 5 was devoted to a compositional method for analysing a call-by-need language, developing from earlier works of Bjerner [Bje89] and Wadler [Wad88]. The central idea in a compositional time-analysis is to parameterise the time-cost of an expression by some abstract description of the context in which the expression is evaluated.

In the first part of this chapter, we showed how information about context in the form of projections satisfying certain properties could be used to define two types of cost-function:

- *sufficient-time* equations which use information about "not-neededness" to give an upper-bound to the time for lazy evaluation.

- *necessary-time* equations giving a dual lower-bound, with better safety properties, which use information about neededness (strictness).

The extension of these techniques to higher-order functions combines the first-order method with a modification of the cost-closure technique, and to a large extent avoids the difficulties of reasoning about "context" in the presence of higher-order functions.

6.2 Further Work

This section summarises some of the possibilities for further work that follow on directly from the developments in this thesis.

Firstly we note the limits of our investigations: we have considered purely sequential models of computation, and we have only considered the time aspects of cost. A formal treatment of parallel evaluation, and space complexity, and the corresponding problems in the context of lazy evaluation, provide many opportunities for further work[1].

The emphasis on expressing program properties as functional programs, seen particularly in **chapter 3**, has been motivated in part by the fact that it enables formal techniques (relatively) well-understood by functional programmers to be used for reasoning about performance. There has been steady interest in the mechanisation of many of these techniques, with a view to providing a machine assisted route to the formal construction of efficient software (for an overview see [PS83]). These programmes suggest possible investigations into the extent to which the specialised program transformation problem of simplifying cost-functions could be tackled by these general systems (rather than by developing a specialist system such as ACE [LeM88b]). For this to be feasible, the transformation system must be extensible to support the various specialised methods which represent procedural knowledge of the problem domain. In a schema-based transformation approach such as [CIP87], this would mean developing appropriate transformation schemata; in more algorithmic (generative set) systems such as [DHK$^+$89], appropriate tactics (meta-programs) would be required.

In the calculus developed in **chapter 4** we proposed a set of time equations together with cost-equivalences based on a theory of cost-simulation. Further investigation is needed to find a suitable set of cost-equivalence laws. A similar programme could be carried out for a higher-order language. Possibilities of adopting this direct approach could be investigated for call-by-need, by manipulating explicit graph structures.

The cost-simulation ideas also raise the possibility of developing other pre-orders involving extensional properties. As an example there is a simple notion of *program refinement*, $\trianglerighteq$. Given two programs p and q, p is refined by q, written $p \trianglerighteq q$ if they

[1]Some considerations of the analysis of space complexity are given in [LeM88a], and Le Métayer's ACE system has been applied to the analysis of FP programs under a parallel reduction model [LeM]. A semantics which models the performance of parallel functional programs has been considered in [Roe90].

are (extensionally) equivalent, and that whenever $p \xrightarrow{H} p'$ then $q \xrightarrow{H} q'$ for some q' such that $p' \trianglerighteq q'$ and $\langle p \rangle^H \geq \langle q \rangle^H$.

The compositional methods in **chapter 5** rely on two separate calculi: one for reasoning about context, in the form of projections, and another to reason about cost. Further work is needed to develop the former calculus. One appeal of the projection-based approach is that we know we are able to obtain *some* safe projections by the application of the automatic techniques from [WH87]. This is appealing, but the degree of approximation in this approach can lead to poor time analyses. This is particularly true in the case of higher-order functions, where we have adopted a less compositional solution using modified cost-closures to partially by-pass these problems; solutions based on a higher-order treatment of context are needed to give improvements to these methods which are less *ad hoc* than those suggested in section 5.5.7.

6.3 Applications

In this section we consider some of the possible areas for further work in the applications of cost-analysis to program transformation and execution on parallel machines.

6.3.1 Program Transformation

Since program transformation promises a formal route to the construction of "efficient" software from clear initial specifications, formal techniques for reasoning about efficiency could provide useful support for this process. An immediate application of cost-analysis techniques is in the comparison of different programs satisfying the same specification, such as pre and post-transformed programs.

Since efficiency issues are so central to the aims of transformational methodologies, the promotion of the rôle of cost-analysis is an interesting area for future work. Wegbreit [Weg76a] has considered the use of cost-analysis for making program transformation more systematic. Wegbreit's goal-directed approach can be summarised as

(i) obtain a lower-bound estimate for the problem's computational cost

(ii) analyse the program to determine the actual cost, relating specific components of cost to specific program "segments"

(iii) determine the segments whose cost is unaccounted for in the minimum cost estimate

(iv) transform the targeted segments

Determining lower bounds on the problem's cost (from some program satisfying that problem) is somewhat difficult, and attempts to locate specific areas in a program where inefficiencies occur ignore the gross structure of the algorithm.

A more useful approach may be one which focuses more on the transformation methods available and uses cost information to guide their application. Developments here may occur at two levels:

Tactic Specific Specific transformation techniques, or tactics, can use information about cost to determine the practicality of their application. For example, *finite differencing* [PK82] aims to replace expensive expressions in a program loop by an incrementally maintained expression. The practicality of this activity depends on the cost of calculation at each iteration relative to the cost of incremental maintenance.

Generalised Approaches An aim of many transformational approaches is to find a general framework for expressing transformation tactics. If we view a transformation tactic T as a mapping between programs, taking some program P to an equivalent one P', then the efficiency-effects of the transformation can be obtained by examining the corresponding cost-programs, CP and CP'. Pictorially:

$$\begin{tikzcd} P \arrow[r, "T"] \arrow[d, Downarrow] & P' \arrow[d, Downarrow] \\ CP \arrow[r, "C_T"] & CP' \end{tikzcd}$$

where the $\Downarrow$ steps represent the derivation of the cost-programs. If, corresponding to the tactic T, the *efficiency-tactic* C_T can be constructed, then it may be possible to provide some of the following:

- cheaper and more effective determination of transformation-effects without having to first perform the transformation

- an improved means of selecting transformations (important since the number of possible transformations may grow exponentially as transformations are composed)

- simpler construction of the final cost-expression CP'

To build this into an extensible transformation-system, it would be desirable to build cost reasoning into the tactics and meta-tactics. In a rule-based system (for example [CIP87]), a *cost-rule* schema could be associated with each rule-schema, which describes how cost changes under the rule; meta-rules which govern the derivation of new rules could then have associated meta-cost-rules which govern the derivation of the dual cost-rules.

These ideas capture the natural move from reasoning about the efficiency of programs, to reasoning about the *meta*-efficiency of meta-programs.

6.3.2 Efficient Parallel Evaluation

Although we have not developed calculi for reasoning directly about the parallel evaluation of functional languages, measurements of sequential cost are particularly relevant in the execution of functional programs on coarse-grain parallel architectures. The efficient implementation of programs on multiprocessors faces the problem of compile-time partitioning of programs into sequential tasks. A crucial factor in the partitioning process is the selection of an appropriate *grain size*—the (average) time-cost of the sequential tasks. With respect to a particular machine, a good choice of grain-size can maximise the utilisation of resources whilst minimising communication overheads.

Much of the literature on the subject of partitioning (and the subsequent job of scheduling) (*e.g.* [Efe82, KL88]) assumes that the cost of processing each module is known, or suggests (*e.g.* [SH86]) that such information be obtained by experimental methods (prototyping with test data). However, the results of more analytic analyses of functional programs can be used to determine a more informed choice of partition. Hudak and Goldberg [HG85] show how a very simplistic cost-analysis can be used to partition a functional program into *serial* combinators, which are intended to be the smallest useful grains of parallelism.

The problem of scheduling—assigning tasks to processors—can also make use of analytic time-information. Maheshwari [Mah89] suggests how complexity information can be used to schedule (functional program) tasks based on their relative costs. The calculi presented in this thesis can enable the programmer, and with some mechanisation perhaps even the compiler, to provide such information.

Bibliography

[Abr90a] S. Abramsky. Abstract interpretation, logical relations and Kan extensions. *Logic and Computation*, 1990. *to appear.*

[Abr90b] S. Abramsky. The lazy lambda calculus. In D. Turner, editor, *Research Topics in Functional Programming.* Addison Wesley, 1990.

[AH87] S. Abramsky and C.L. Hankin, editors. *Abstract Interpretation of Declarative Languages.* Ellis Horwood, 1987.

[AHU74] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms.* The Addison-Wesley Series in Computer Science and Information Processing. Addison-Wesley Publishing Company, London, 1974.

[AHU82] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *Data Structures and Algorithms.* The Addison-Wesley Series in Computer Science and Information Processing. Addison-Wesley Publishing Company, London, 1982.

[Ast89] E. Astesiano. Operational semantics. In *IFIP working group 2.2*, Rio de Janero, 1989.

[Bac78] J. Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, August 1978.

[Bar84] H.P. Barendregt. *The Lambda Calculus*, volume 103 of *Studies in Logic and the Foundations of Mathematics.* Elsevier Science Publishers B.V., P.O. Box 1991, 1000 BZ Amsterdam, The Netherlands, 2nd edition, 1984.

[BD77] R. Burstall and J. Darlington. A transformation system for developing recursive programs. *JACM*, 24:44–67, January 1977.

[BH87] R. Burstall and F. Honsell. A natural deduction treatment of operational semantics. In *Proceedings of the* 8^{th} *Conference on the Foundations of Software Technology*, number 287 in LNCS, pages 250–269. Springer Verlag, 1987.

[BH88] A. Bloss and P. Hudak. Path semantics. In *The Third Workshop on the Mathematical Foundations of Programming Language Semantics.* Springer Verlag, 1988.

[BH89] B. Bjerner and S. Holmström. A compositional approach to time analysis of first order lazy functional programs. In *Functional Programming Languages and Computer Architecture, conference proceedings.* ACM press, 1989.

[Bir84] R. S. Bird. The promotion and accumulation strategies in transformational programming. *ACM ToPLaS*, 6:487–504, October 1984.

[Bir86] R. J. Bird. An introduction to the theory of lists. Technical report, Programming Research Group, University of Oxford, 1986.

[Bje89] B. Bjerner. *Time Complexity of Programs in Type Theory.* PhD thesis, Chalmers University of Technology, 1989.

[Bur90] G.L. Burn. A relationship between abstract interpretation and projection analysis (extended abstract). In *17th ACM Symposium on Principles of Programming Languages*, January 1990.

[BvEG+87] H.P. Barendregt, M.C.J.D. van Eekelen, J.R.W. Glauert, J.R. Kennaway, M.J. Plasmeijer, and M.R. Sleep. Term graph rewriting. In *PARLE '87 volume II*, number 259 in LNCS, pages 191–231. Springer Verlag, 1987.

[BW88] R. Bird and P. Wadler. *Introduction to Functional Programming.* Prentice Hall, 1988.

[Chi90] W. N. Chin. *Automatic Methods for Program Transformation.* PhD thesis, Imperial College, University of London, 1990.

[CIP87] Systems Group CIP. *The Munich Project CIP, Volume II: The program transformation system.* Number 292 in LNCS. Springer-Verlag, 1987.

[CK77] J. Cohen and J. Katcoff. Symbolic solution of finite-difference equations. *Transactions on Mathematical Software*, 3:261–271, September 1977.

[Coh82] J. Cohen. Computer-assisted microanalysis of programs. *C. ACM*, 25:44–67, October 1982.

[DHK+89] J. Darlington, P. Harrison, H. Khoshnevisan, L. McLoughlin, N. Perry, H. Pull, M. Reeve, K. Sephton, L. While, and S. Wright. A functional programming environment supporting execution, partial execution and transformation. In *PARLE '89, Parallel Architectures and Languages Europe*, number 365 in LNCS. Springer Verlag, June 1989. Volume 1: Parallel Architectures.

[DM82] L. Damas and R. Milner. Principal type schemes for functional programs. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 207–212, 1982.

[DW89] K. Davis and P. Wadler. Backwards strictness analysis: Proved and improved. In *Proceedings of Glasgow Workshop on Functional Programming*, Workshop series. Springer-Verlag, August 1989.

[Efe82] K. Efe. Heuristic models of task assignment scheduling in distributed systems. *IEEE Computer*, 15(6):50–56, June 1982.

[Fea82] M. S. Feather. A system for assisting program transformation. *IEEE Transactions on Software Engineering*, 8:490–498, September 1982.

[FH88] A. J. Field and P. G. Harrison. *Functional Programming*. Addison-Wesley, 1988.

[Fla85] P. Flajolet. Mathematical methods in the analysis of algorithms and data structures. Rapport 400, INRIA, Le Chesnay, France, May 1985.

[FS81] P. Flajolet and J-M Steyaert. A complexity calculus for classes of recursive search programs. In *Proceedings of th 22nd Annual Symposium on Foundations of Computer Science*, pages 386–393. IEEE press, 1981.

[GKP89] R. L. Graham, D. E. Knuth, and O. Patashnik. *Concrete Mathematics*. Addison-Wesley, 1989.

[HC88] T. Hickey and J. Cohen. Automating program analysis. *J. ACM*, 35:185–220, January 1988.

[HG85] P. Hudak and B. Goldberg. Serial combinators: "optimal grains" of parallelism. In *Symposium of Functional Programming Languages and Computer Architecture*, 1985.

[HM76] P. Henderson and J.M. Morris. A lazy evaluator. In *Proceedings of the Third POPL Symposium*, pages 95–103, Atlanta Georgia, January 1976.

[HMT88] R. Harper, R. Milner, and M. Tofte. The definition of standard ML (version 2). Technical Report ECS-LFCS-88-62, LFCS, Department of Computer Science, University of Edinburgh, The King's Buildings, Edinburgh EH9 3JZ, UK, August 1988.

[Hud87] P. Hudak. A semantic model of reference counting and its abstraction, 1987. In [AH87].

[Hud89] P. Hudak. Conception, evolution and application of functional programming languages. *Computing Surveys*, 21(3):359–411, September 1989.

[Hug87] R. J. M. Hughes. Backwards analysis of functional programs. Research Report CSC/87/R3, University of Glasgow, March 1987.

[Hug88] R. J. M. Hughes. Abstract interpretation of first–order polymorphic functions. In *Glasgow Workshop on Functional Programming*, University of Glasgow, Department of Computing Science, August 1988. Research Report 89/R4.

[Hug89] J. Hughes. Why functional programming matters. *The Computer Journal*, 2(32):98–107, April 1989.

[Hun90a] S. Hunt. PERs generalise projections for strictness analysis. In *Draft Proceedings of the Third Glasgow Functional Programming Workshop*, Ullapool, 1990.

[Hun90b] S. Hunt. Projection analysis and stable functions. Draft paper, Imperial College, 1990.

[HW89] C. V. Hall and D. S. Wise. Generating function versions using rational strictness patterns. *Science of Computer Programming*, 12:39–74, 1989.

[HY86] P. Hudak and J. Young. Higher-order strictness analysis in untyped lambda calculus. In *Proceedings of the 13th Annual Symposium on Principles of Programming Languages.* ACM SIGPLAN, 1986.

[Jen90] T. P. Jensen. Context analysis of functional programs. Master's thesis, DIKU, Copenhagen, Denmark, 1990.

[JM89a] S. B. Jones and D. Le Métayer. Compile-time garbage collection by sharing analysis. In *Functional Programming Languages and Computer Architecture, conference proceedings*, pages 54–74. ACM press, 1989.

[JM89b] S. B. Jones and D. Le Métayer. A new method for strictness analysis. In *Proceedings of Glasgow Workshop on Functional Programming*, Workshop series. Springer-Verlag, August 1989.

[Kah87] G. Kahn. Natural semantics. In *Proceedings of Symposium on Theoretical Aspects of Computer Science*, pages 22–39. Springer Verlag, 1987. LNCS 247.

[KL88] B. Kuatrachue and T. Lewis. Grain size determination for parallel processing. *IEEE Software*, 5(1):23–32, January 1988.

[Klo80] J.W. Klop. *Combinatory Reduction Systems*, volume 127 of *Mathematical Centre Tracts*. Mathematischen Centrum, 413 Kruislaan, Amsterdam, 1980.

[Knu68] D. E. Knuth. *Volume 1: Fundamental Algorithms*. The Art of Computer Programming. Addison-Wesley, 1968.

[Knu69] D. E. Knuth. *Volume 2: Seminumerical Algorithms*. The Art of Computer Programming. Addison-Wesley, 1969.

[Knu73] D. E. Knuth. *Volume 3: Sorting and Searching*. The Art of Computer Programming. Addison-Wesley, 1973.

[Lan64] P. J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6(4):308–320, January 1964.

[Lau89] J. Launchbury. *Projection Factorisations in Partial Evaluation*. PhD thesis, Department of Computing, University of Glasgow, 1989.

[LeM] D. LeMétayer. Personal communication.

[LeM85] D. LeMétayer. Mechanical analysis of program complexity. In *ACM SIGPLAN 85 Symposium*, July 1985.

[LeM88a] D. LeMétayer. Analysis of functional programs by program transformation. In *Second France–Japan Artificial Intelligence and Computer Science Symposium.* North–Holland, 1988.

[LeM88b] D. LeMétayer. An automatic complexity evaluator. *ACM ToPLaS*, 10(2):248–266, April 1988.

[Mah89] P. Maheshwari. Efficient parallel execution of functional programs using complexity information. Draft article, University of Manchester, October 1989.

[McC67] J. McCarthy. *Computer Programming and Formal Systems*, chapter A Basis for a Mathematical Theory of Computation. North–Holland, 1967.

[Mee86] L. Meertens. Algorithmics. In M Hazewinkel J. W. de Bakker and L. K. Lenstra, editors, *Mathematics and Computer Science*, volume I of *CWI monographs*, pages 289–334. North Holland, 1986.

[Mil83] R. Milner. Calculi for synchrony and asynchrony. *Theoretical Computer Science*, 25:267–310, 1983.

[Mil89] R. Milner. *Communication and Concurrency.* Prentice Hall, 1989.

[Mog88] T. Æ. Mogensen. Binding time analysis for higher order polymorphically typed languages. Technical report, DIKU, August 1988.

[Nei84] H. R. Neilson. *Hoare Logic's for Run-time Analysis of Programs.* PhD thesis, Department of Computer Science, Edinburgh, 1984.

[O'D77] M. J. O'Donnell. *Computing in systems described by equations*, volume 58 of *LNCS.* Springer Verlag, 1977.

[Par80] D. Park. Concurrency and automata on infinite sequences. In *5th GI conference on Theoretical Computer Science.* LNCS 104, Springer Verlag, 1980.

[Per88] N. Perry. Hope+. Functional programming group internal report IC/FPR/LANG/2.5.1/7, Department of Computer Science, Imperial College, February 1988.

[Pey87] S. L. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice-Hall International Series in Computer Science. Prentice-Hall International (UK) Ltd, London, 1987.

[PK82] R. Paige and S. Koenig. Finite differencing of computible expressions. *ACM ToPLaS*, 4(3):402–454, July 1982.

[Plo75] G. D. Plotkin. Call-by-name, Call-by-value and the λ-calculus. *Theoretical Computer Science*, 1(1):125–159, 1975.

[Plo81] G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN–19, Computer Science Department, Aahus University, Denmark, September 1981.

[PS83] P. Partsch and R. Steinbruggen. Program transformation systems. *Computing Surveys*, 15:199–236, 1983.

[Ram79] L. H. Ramshaw. Formalizing the analysis of algorithms. Technical Report CSL–79–5, XEROX Palo Alto Research Center, June 1979.

[Rey72] J. C. Reynolds. Definitional interpreters for higher order programming languages. In *Proceedings of the 25th ACM National Conference*, pages 717–740, Boston, August 1972.

[Roe90] P. Roe. A semantics for reasoning about parallel programs' performance. In *Draft Proceedings of the third Glasgow functional programming group workshop*, 1990.

[Ros86] M. Rosendahl. Automatic construction of time bound programs. Master's thesis, University of Copenhagen, 1986.

[Ros89] M. Rosendahl. Automatic complexity analysis. In *Functional Programming Languages and computer architecture, conference proceedings.* ACM press, 1989.

[San89] D. Sands. Complexity analysis for a lazy higher-order language. In *Proceedings of Glasgow Workshop on Functional Programming*, Workshop Series. Springer Verlag, August 1989.

[San90] D. Sands. Complexity analysis for a lazy higher-order language. In *Proceedings of the Third European Symposium on Programming*, number 432 in LNCS. Springer-Verlag, May 1990.

[Sch86] D. A. Schmidt. *Denotational Semantics.* Allyn and Bacon, Inc., Massachusetts, 1986.

[Ses89] P. Sestoft. Replacing function parameters by global variables. In *Functional Programming Languages and Computer Architecture, London, September 89.* ACM Press and Addison-Wesley, 1989.

[SH86] V. Sarkar and J. Hennessey. Compile-time partitioning and scheduling of parallel programs. In *Proceedings of the ACM SIGPLAN Symposium on Compiler Construction*, pages 17–26, 1986.

[Shu85] J. Shultis. On the complexity of higher-order programs. Technical Report CU-CS-288, University of Colorado, Febuary 1985.

[SW71] C. Strachey and C. P. Wadsworth. Continuations – a mathematical sematics for handling full jumps. Technical Monograph PRG-11, Oxford, 1971.

[Tal85a] C. L. Talcott. Derived properties and derived programs. Technical report, Stanford, 1985.

[Tal85b] C. L. Talcott. *The Essence of Rum, A Theory of the intensional and extensional aspects of Lisp-type computation.* PhD thesis, Stanford University, August 1985.

[Tur81] D.A. Turner. The semantic elegance of applicative languages. In *Proceedings of the 1981 Conference on Functional Programming Languages and Computer Architecture*, pages 82–95. ACM, 1981.

[Wad71] C. P. Wadsworth. *Semantics and Pragmatics of The Lambda Calculus.* Dphil thesis, University of Oxford, 1971.

[Wad88] P. Wadler. Strictness analysis aids time analysis. In *15th ACM Symposium on Principals of Programming Languages*, January 1988.

[Weg75] B. Wegbreit. Mechanical program analysis. *C.ACM*, 18:528–539, September 1975.

[Weg76a] B. Wegbreit. Goal–directed program transformation. *IEEE Transactions on Sofware Engineering*, 2:69–80, June 1976.

[Weg76b] B. Wegbreit. Verifying program performance. *J.ACM*, 23:691–699, 1976.

[WH87] P. Wadler and R. J. M. Hughes. Projections for strictness analysis. In *1987 Conference on Functional Programming and Computer Architecture*, Portland, Oregon, September 1987.

[Wra86] S. C. Wray. Programming techniques for functional languages. Technical Report 92, University of Cambridge Computer Laboratory, June 1986.

[ZZ89] P. Zimmermann and W. Zimmermann. The automatic complexity analysis of divide-and-conquer algorithms. Technical Report Rapports de Recherche 1149, INRIA-Rocquencourt, Décembre 1989.