A Functional Semantics for Space and Time

by Catherine Hope, BSc (Hons)

Thesis submitted to The University of Nottingham
for the degree of Doctor of Philosophy, 18th May 2008

Contents

Abstract

In this thesis we focus on the problem of reasoning about the space and time usage of functional programs. Such reasoning is less straightforward than for imperative programs, due to the lack of explicit control flow and memory management that is inherent in the functional style. Our approach to making such information explicit is to consider the behaviour of an underlying abstract machine for executing functional programs, at which level both control flow, and the additional data structures required for execution, are manifest. However, we do not wish to reason about space and time usage at this low-level, but rather we seek a way of reflecting this information back to the level of the original program. Moreover we aim to do this in a calculational manner, by deriving space and time information using a process of systematic equational reasoning.

The primary contribution of the thesis is an approach to achieving our aim that proceeds as follows. Starting with an evaluator that expresses the intended semantics of the language at a high-level, we derive an abstract machine by a systematic process of calculation. This machine is then instrumented with space or time information, and the same calculational process applied in reverse to obtain a high-level instrumented evaluator. The resulting evaluator can then be specialised for a particular program and, finally, transformed into a closed form expression for space or time usage. As well as presenting a novel approach to obtaining this information, the thesis also provides a new application of a number of established program transformation techniques, such as defunctionalization and transformation into continuation-passing style. We demonstrate the practicality of our technique with a range of programming examples, showing how to derive simple expressions that accurately reflect the space and time behaviour of an underlying abstract machine. We also present a new deforestation theorem, based upon a novel form of hylomorphism, and show how our techniques can be applied to formalise its space performance.

Chapter 1

Background

1.1 Motivation

Due to ever-faster processors and cheaper memory, the resource requirements of programs, both in terms of execution time and memory usage, may not seem as important as in the past. However, these hardware advances are coupled with more complex and resource-demanding software. Combined with the increasing ubiquity of embedded systems, often having tight resource bounds, this means that there will always be a demand for accurate space and time information about programs. Estimating resource consumption of a program, that is the amount of space and time required for execution, is therefore useful for two main reasons: it can be used to compare two programs that solve the same problem, in order to chose the most efficient version, or to verify that a program will run successfully given space or time constraints.

How are these resource requirements to be produced? Naively, we might run a program on a range of inputs and time how long it takes to finish, and measure the memory allocations for space usage. However, this measure would be dependent on many issues, such as the choice of inputs, the hardware being used, and the compiler. Such concerns motivate the development of higher-level approaches to measuring resource requirements, based upon static analysis of the source code of a program.

Producing these metrics for traditional, imperative, programming languages, where execution order is explicit, is a matter of summing time and space requirements di-

rectly in a pass through the program code. For a program in a functional language, however, there is no explicit sequence of instructions, and the allocation of new memory is performed transparently, so producing this information is more complicated.

One of the strengths of functional languages is the ability to prove properties and verify correctness of programs. Much work has been done on reasoning about these so-called *extensional* properties of functional languages, but if these languages are to move beyond being mainly research-based, being able to prove properties about the space and time performance of programs will also be necessary. Such properties are called *intensional*, in that they come from the inner workings of execution rather than than directly from the result of running a program.

A common issue for program analysis in imperative languages is what to count as an atomic unit of execution. In the imperative world it is common practice to count variable lookups and variable assignments, for example. By contrast, functional programs are evaluated by performing a series of reductions, until the program is reduced to a value. The traditional approach to measuring time performance is to use these reductions as the atomic units. However, each of these reductions may take arbitrarily long, so this measure may not be very accurate.

An alternative atomic unit would be to measure transitions of an underlying abstract machine. We suggest that this is a more realistic measure, since it models how a program would execute on an actual machine. The motivation for this work is the recent developments by Danvy *et al* in *deriving* abstract machines from high-level evaluators, by applying successive program transformations. Here we will show how to *calculate* machines, by forming a specification and applying equational reasoning techniques in order to produce the definition of the function [1]. The essential difference here is one of methodology: Danvy focuses on applying known techniques to reach a goal, whereas we focus on systematically discovering the goal. At this low-level the machine is instrumented with space and time information, and then an evaluator is recalculated, using the same process but in the reverse order. The result is a high-level function that accurately measures the time behaviour of programs. A similar process can be performed for measuring space usage, by instead counting the

size of the additional data structures created at the abstract machine level. In this way we can get more accurate space and time information about evaluation.

In the next section, we will first consider how space and time analysis is performed for imperative languages, and then see how this can be adapted to functional languages. Functional examples are presented in the syntax of Haskell [2], but we will use different evaluation strategies according to the context.

1.2 Imperative approach

Programs in an imperative language have an explicit sequence of steps to follow to produce a result. Analysis is performed by first choosing what to count as an atomic, or basic, operation. This must be an operation whose execution time, or occupying space, can be assumed to be independent of the values of its arguments. For example, the addition of two numbers could be taken as an atomic unit if it always took the same amount of time to perform, independent of the value of its numerical arguments.

As the execution order is explicit in the program, the cost of the whole program is the sum of the cost of each line. Considering the following imperative code fragment, the execution behaviour depends on the value of the variable n:

```
int sum=0;                    (1)
int i=0;                      (1)
while (i<n) {                 (n)
  sum = sum + array[i];       (4 * n)
  i = i + 1;                  (3 * n)
}
```

The cost for each line of the program is calculated by taking assignment, array lookup, comparison and addition as atomic units of operation for execution. These costs are annotated on the right of the code, and summing these amounts together gives the total execution cost of $8 * n + 2$. In the while-loop, the costs are multiplied by the number of times the loop is repeated, in this case n. If we were interested in the

space requirements, then we could take the declaration of variables holding integers as atomic, and the requirements would be simply 2 units, since two integer variables are created during execution.

The most accurate choice of atomic unit would be to count machine-code level instructions, which are directly executed. At this low-level perspective, the program state is defined by the contents of the memory, and the statements are instructions in the native machine language of the computer. However, this would be too low-level to relate back to the original program, and also too machine specific — a measure that would be relevant to any hardware would be preferable.

Any choice of atomic unit (provided it satisfied the constraint of not depending on the value of its arguments) would still be an accurate measure since, when compiled, each high-level construct in an imperative language is translated into one or more machine code instructions, which are directly executed. If the granularity of the atomic unit does not matter, then the next step is to abstract away from any particular choice and instead concentrate on the growth rate of the algorithm. This can be deduced from the costs produced from the atomic units by taking the part of the result with the largest growth rate. For example, the above code would be linear for execution time, and have constant space requirements.

Asymptotic analysis is used to describe the growth rate of an algorithm for large inputs [3], above which the overall rate of growth becomes apparent. A common asymptotic measure is O complexity, which gives an upper bound on the rate of growth. The reasoning is that for large input values any constant factors will become less important, and the sub-expression with the steepest growth will have the major effect on the behaviour. Formally, this idea can be expressed as follows:

$$\forall n > n_0 . T(f(n)) \leq cg(n) \;\Rightarrow\; T(f(n)) \in O(g(n))$$

This property states that for all inputs n greater than a constant n_0, if the program f called on n always executes in less time than a constant factor c multiplied by a function g applied to n, then g gives an upper bound on the execution time. If this is

the case then the time requirements for $f(n)$, expressed as $T(f(n))$, have complexity $O(g(n))$. The constant n_0 places a lower bound on the range of input values to consider, while c abstracts out any constant factors.

There are other complexity measures, such as θ and ω notation, which give exact and lower bounds respectively. Using asymptotic analysis removes any constants involved, but this loss of accuracy may be an important consideration for smaller input sizes (under the n_0 value), as well as for applications with tight resource bounds.

1.3 Functional approach

Unlike imperative languages, programs in a pure functional language are not a sequence of statements, and have no variable assignments or explicit memory allocation at the program level, so performing space and time analysis is more complicated. However, an important benefit of the lack of global state is that it considerably simplifies the process of reasoning about programs, allowing the use of standard equational reasoning techniques [4]. In this section we briefly review a number of concepts concerning the evaluation of functional programs.

Evaluation

A program in a functional language consists of a set of named definitions, which are used as rewrite rules. Programs are *evaluated* by performing successive reductions on expressions until they are in *normal form*, where they can no longer be further reduced. A reduction occurs by first isolating a reducible sub-expression, a *redex*, which matches the left hand side of a rewrite rule, and then replacing it with the corresponding right-hand side. The choice of redex is specified by an evaluation strategy, which may have an effect on time and space performance.

Evaluation strategies

Two common evaluation strategies are call-by-value and call-by-name, which respectively select the inner and outer-most redexes to reduce. An innermost redex is one that does not contain a redex, and similarly an outermost redex is one that is not contained in a redex. For example, the arithmetic expression $(1 + 2)^2$ may be evaluated by either choosing the innermost redex $(1 + 2)$ to reduce first, or performing the outermost exponentiation redex, as illustrated below:

Call-by-value		Call-by-name
$(1 + 2)^2$		$(1 + 2)^2$
$= 3^2$	$=$	$(1 + 2) * (1 + 2)$
$= 3 * 3$	$=$	$3 * (1 + 2)$
$= 9$	$=$	$3 * 3$
	$=$	9

In the call-by-value strategy the evaluation of the inner redex $(1+2)$ is performed first, before passing the result to the exponentiation function. This copies its arguments and then multiplies them together to produce the result 9. By contrast, in the call-by-name strategy, the definition of the exponentiation function is applied first, copying the sub-expression $(1+2)$, and then the evaluation of the addition has to occur twice, because the unevaluated expression has been duplicated. Once this has occurred, the multiplication can be performed to give the same result 9. Note that in the second line of the call-by-name trace, $(1 + 2) * (1 + 2)$, there is a choice of redex to reduce, and, by convention, the leftmost one is chosen, as this corresponds to the normal left-to-right evaluation of the arguments of a function.

The time requirements for evaluation can be informally measured by using the length of the evaluation trace. In the case of the call-by-value strategy, 3 reductions are required in order to reach the result, whereas the call-by-value strategy requires an additional step, corresponding to the repeated evaluation of $(1 + 2)$.

The space requirements can also be estimated using the trace. In the definition of

exponentiation the argument is first copied, and then multiplied. In the call-by-name strategy a larger sub-expression, $(1 + 2)$, has to be copied than in the call-by-value case, where the argument is the integer 3, so call-by-name has greater space requirements in this case. This simple trace illustrates an important difference between both approaches, in that although evaluating with different strategies produces the same result, the time and space requirements for doing so may differ.

An expression evaluated under call-by-value will often require less time and space than under call-by-name because it can avoid repeatedly evaluating common sub-expressions, but sometimes the opposite is true. Consider the function *head* that returns the first element of a list, and the generator $[1\,..\,100]$ that creates a list from 1 to 100. Evaluating the expression *head* $[1\,..\,100]$ under call-by-value causes the list to be completely expanded before the application of the *head* definition, resulting in both linear time and space requirements. By contrast, the call-by-name strategy evaluates this expression in both constant time and space, because the *head* function can be applied as soon as the first element is generated, as shown below:

	Call-by-value		Call-by-name
	head $[1\,..\,100]$		*head* $[1\,..\,100]$
$=$	*head* $(1 : [2\,..\,100])$	$=$	*head* $(1 : [2\,..\,100])$
$=$	*head* $(1 : 2 : [3\,..\,100])$	$=$	1
$=$	$\vdots$		
$=$	1		

The same is true in any case where the result of a sub-expression is not required: call-by-name could avoid having to evaluate it, and consequently execute in less time and less space. An issue relating to when this can happen is that of strictness.

Strictness

A function is described as *strict* if it requires the value of its argument. For example, if the boolean conjunction function $\wedge$ is defined as follows,

$$(\wedge) :: Bool \rightarrow Bool \rightarrow Bool$$
$$False \ \wedge \ y = False$$
$$True \ \wedge \ y = y$$

then this function is strict in its first argument, since it must be fully evaluated to perform pattern matching. However, if the first argument is *False* then it does not need to look at the second, so it is non-strict in its second argument. Formally a function f that takes a single argument is strict if $f\bot = \bot$, where $\bot$ represents the undefined value. In turn, a (curried) function g with two arguments is strict in its first argument if $g \ \bot \ y = \bot$ for any value y, and similarly for its second argument. Hence, $\wedge$ is not strict in its second argument, because $False \wedge \bot \neq \bot$. In terms of execution time, this also means that call-by-name evaluation will be more efficient in evaluating $False \ \wedge \ y$, because evaluation of the expression y is not required.

Normal forms

As illustrated in the trace for $(1 + 2)^2$, if two evaluation strategies executing an expression produce a result in normal form, then they will be equal. This is formalised in the well-known (first) Church-Rosser theorem, which states that if an expression can be reduced to two distinct expressions, then both of these may be further reduced to the same expression. A consequence of this is that if there exists a normal form for an expression (this is not always possible because reduction sequences may be non-terminating) then it is unique. However, as shown in the earlier trace, this does not mean that the intensional requirements of evaluation are the same, since the number of reduction rules applied may vary, and the order they are applied in, as well as the size of the sub-expressions duplicated, will affect how much space is required.

Non-termination

An advantage of call-by-name over call-by-value is that any expression can be reduced to its normal form, if it exists, by using the call-by-name evaluation strategy. Technically, this is known as the second Church-Rosser theorem. If only expressions

which contribute to the final result are evaluated, then in addition to increased efficiency, one also gains programming advantages, such as being able to work with infinite sized data structures (which are only evaluated as much as required by the context in which they are used) and non-strict functions (whose arguments are, once again, only evaluated where necessary).

Consider an infinite stream of 1's, defined by:

$$ones = 1 : ones$$

A simple example of call-by-value evaluation causing non-termination occurs when evaluating the expression *head ones*:

	Call-by-value		Call-by-name
	head ones		*head ones*
=	*head* $(1 : ones)$	=	*head* $(1 : ones)$
=	*head* $(1 : 1 : ones)$	=	1
	$\vdots$		

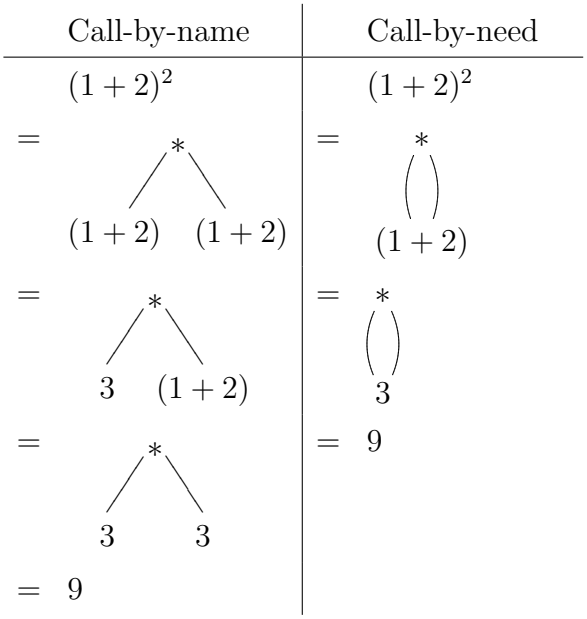
In this example, call-by-value will always choose the innermost redex and repeatedly apply the *ones* definition. In this case, however, the reduction sequence will never terminate. By comparison, under call-by-name evaluation the definition of *ones* need only be unwound once to produce a list constructor : that matches the left-hand side of the *head* definition, which reduces to the result 1.

Graph reduction

The benefits of one-time evaluation of sub-expressions that occur under call-by-value can also be achieved in call-by-name by introducing sharing. Usually an expression in a functional language is viewed as a tree, with function application at the nodes and arguments as children. However in practice expressions in modern functional languages such as Haskell are represented as graphs in which common sub-expressions are shared. This provides both space and time benefits, in that equal sub-expressions are not duplicated as they are in the usual tree representation, and are only evaluated

once, with the resulting value being re-used if required later on. In fact, shared sub-expressions are evaluated *at most* once, because the outermost strategy only evaluates an expression if its result is required. The combination of call-by-name with graph sharing in this manner is called *call-by-need* or *lazy* evaluation.

The trace below shows the difference in evaluation between call-by-name, where the expression is modeled as a tree, and call-by-need, where it is a graph:



The arguments to functions are still passed in unevaluated except that in the call-by-need case, instead of being duplicated, a reference to the sub-expression is used. This means that the sub-expression $(1 + 2)$ only needs to be evaluated once.

None of the programming examples that we will consider in this thesis rely on sharing, and so for simplicity we instead choose to focus our attention on the call-by-value and call-by-name evaluation strategies. We will consider the impact of sharing on the ability to reason about programs when we review related work at the end of this chapter, and return to the issue of how our approach may be extended to take account of sharing in chapter 7.

1.4 Examples

As an introduction to reasoning about the space and time usage, some simple examples will be presented in Haskell, and their evaluation traces analysed.

Summing a list

As a first example, we consider the function to sum a list of integers:

$$
\begin{aligned}
&sum && :: [Int] \to Int \\
&sum\ [\,] && = 0 \\
&sum\ (x : xs) && = x + sum\ xs
\end{aligned}
$$

The reduction sequence below shows how the *sum* function applied to the list $[1, 2, 3]$ would be evaluated under the call-by-value strategy (and also for call-by-name since the argument is already fully evaluated):

	Expression	Space profile
	$sum\ (1 : (2 : (3 : [\,])))$	□□□□□□□□
$=$	$1 + sum\ (2 : (3 : [\,]))$	□□□□□□□□
$=$	$1 + (2 + sum\ (3 : [\,]))$	□□□□□□□□
$=$	$1 + (2 + (3 + sum\ [\,]))$	□□□□□□□□
$=$	$1 + (2 + (3 + 0))$	□□□□□□□
$=$	$1 + (2 + 3)$	□□□□□
$=$	$1 + 5$	□□□
$=$	6	□
	Time steps = 7	Space units = 8

Estimating time

The amount of time to evaluate an expression can be estimated by the length of the evaluation trace. By observing the evaluation trace above, the number of steps required to calculate the sum of a list of n items can be deduced to be $1 + 2 * n$.

This formula arises because each : operator has to be replaced by addition (n steps), followed by the empty list [] by zero (1 step), and finally each addition has to be performed (n steps). We conclude therefore that for a fully evaluated argument list, the function *sum* has linear time performance, because the time required to evaluate *sum* is proportional to the size of its argument, which is the length of the list.

Estimating space

The space profile on the right of the trace above represents the space the expression is currently occupying. This is estimated by counting the constructors in the expression. During evaluation, the expression may grow and shrink as reduction rules are applied. If space can be re-used in the next step of evaluation, then the amount of space required will be the maximum expression size built during evaluation. This is known as the *residency* of the computation. Another possible measure is the total space required for evaluation, calculated by simply summing up the expression size at each stage of evaluation. Ideally memory can be re-used at each step of the evaluation through the use of a garbage collector, so we choose to take the residency, that is the maximum expression size produced, as the space requirements for evaluation. In practice, garbage collection may involve a significant time overhead, but reasoning about the behaviour of garbage collection algorithms is a subject for a thesis in its own right, and beyond the scope of this present work.

The size of an expression can be approximated by counting the number of values, constructors and operators. This is represented by the boxes in the space profile of the evaluation trace. For example, the size of the expression $1 + sum\ (2 : (3 : [\,]))$ is eight units, which is the sum of three integers, two list constructors and the empty list, and the addition and sum function. The space required to evaluate *sum xs* is constant, assuming the list *xs* is already evaluated and the space for the list is already allocated, because the maximum expression size is one more than the size of the list. The space profile shows that the expression size stays constant while each : is replaced by +, and then decreases as each addition is performed.

Accumulating sum

As a second example, let us consider an alternative definition for the *sum* function, in which an accumulator is used:

$$
\begin{aligned}
&\mathit{sumAcc} && :: [\mathit{Int}] \to \mathit{Int} \\
&\mathit{sumAcc} && = \mathit{sum}'\ 0 \\
&\mathit{sum}' && :: \mathit{Int} \to [\mathit{Int}] \to \mathit{Int} \\
&\mathit{sum}'\ a\ [\,] && = a \\
&\mathit{sum}'\ a\ (x : xs) && = \mathit{sum}'\ (a + x)\ xs
\end{aligned}
$$

This differs from the previous definition in that an extra argument is passed to the function, which is used to keep a running total of the sum. This doesn't affect the number of steps required, apart from one additional step in this implementation to call the auxiliary function *sum′*, but does have an effect on the shape of the space profile, since the addition can now be performed as the list is processed. This is illustrated in the trace below, using call-by-value evaluation:

	Expression	Space profile
	$\mathit{sumAcc}\ (1 : (2 : (3 : [\,])))$	□□□□□□□□
=	$\mathit{sum}'\ 0\ (1 : (2 : (3 : [\,])))$	□□□□□□□□□
=	$\mathit{sum}'\ (0 + 1)\ (2 : (3 : [\,]))$	□□□□□□□□□
=	$\mathit{sum}'\ 1\ (2 : (3 : [\,]))$	□□□□□□□
=	$\mathit{sum}'\ (1 + 2)\ (3 : [\,])$	□□□□□□□
=	$\mathit{sum}'\ 3\ (3 : [\,])$	□□□□□
=	$\mathit{sum}'\ (3 + 3)\ [\,]$	□□□□□
=	$\mathit{sum}'\ 6\ [\,]$	□□□
=	6	□
	Time steps = 8	Space units = 9

If the expression $\mathit{sumAcc}\ [1, 2, 3]$ is evaluated using the call-by-name strategy, then the addition isn't performed until the list has been completely processed, because it is not the outermost redex. So it gives a similar space profile to the original definition

of *sum*:

	Expression	Space Profile
	$sumAcc\ (1 : (2 : (3 : [\,])))$	□□□□□□□□
=	$sum'\ 0\ (1 : (2 : (3 : [\,])))$	□□□□□□□□□
=	$sum'\ (0 + 1)\ (2 : (3 : [\,]))$	□□□□□□□□□
=	$sum'\ ((0 + 1) + 2)\ (3 : [\,])$	□□□□□□□□□
=	$sum'\ (((0 + 1) + 2) + 3)\ [\,]$	□□□□□□□□□
=	$((0 + 1) + 2) + 3$	□□□□□□□
=	$(1 + 2) + 3$	□□□□□
=	$3 + 3$	□□□
=	6	□
	Time steps = 8	Space units = 9

This demonstrates that the space behaviour of a function depends on how the function is defined as well as which evaluation strategy is used. We now consider an example of how the time characteristics of functions can be improved using similar techniques.

Reversing a list

The function to reverse a list can be defined as follows:

$$
\begin{aligned}
&reverse && :: [a] \to [a] \\
&reverse\ [\,] && = [\,] \\
&reverse\ (x : xs) && = reverse\ xs +\!\!+ [x]
\end{aligned}
$$

The append function $+\!\!+$ is defined by recursion on its first argument:

$$
\begin{aligned}
&(+\!\!+) && :: [a] \to [a] \to [a] \\
&[\,] && +\!\!+ ys = ys \\
&(x : xs) && +\!\!+ ys = x : (xs +\!\!+ ys)
\end{aligned}
$$

Evaluating *reverse* $[1,2,3]$ using the call-by-value and call-by-name strategies gives the same trace, since they both choose the same redex to perform at each stage:

	Expression	Space Profile
	$reverse\ (1:(2:(3:[\,])))$	□□□□□□□□
$=$	$reverse\ (2:(3:[\,]))+\!\!+(1:[\,])$	□□□□□□□□□
$=$	$(reverse\ (3:[\,])+\!\!+(2:[\,]))+\!\!+(1:[\,])$	□□□□□□□□□□
$=$	$((reverse\ [\,]+\!\!+(3:[\,]))+\!\!+(2:[\,]))+\!\!+(1:[\,])$	□□□□□□□□□□□
$=$	$(([\,]+\!\!+(3:[\,]))+\!\!+(2:[\,]))+\!\!+(1:[\,])$	□□□□□□□□□□□
$=$	$((3:[\,])+\!\!+(2:[\,]))+\!\!+(1:[\,])$	□□□□□□□□□□
$=$	$3:([\,]+\!\!+(2:[\,]))+\!\!+(1:[\,])$	□□□□□□□□□□
$=$	$(3:2:[\,])+\!\!+(1:[\,])$	□□□□□□□□
$=$	$3:((2:[\,])+\!\!+(1:[\,]))$	□□□□□□□□
$=$	$3:(2:([\,]+\!\!+(1:[\,])))$	□□□□□□□□
$=$	$3:(2:(1:[\,]))$	□□□□□□□
	Time steps = 10	Space units = 14

From the above profile, we can see that space can be saved if we could perform a running append instead of having to wait until the end of the input list is reached.

Fast reverse

The desired space improvement can be obtained by redefining the *reverse* function using an accumulator:

$$
\begin{aligned}
&reverseAcc &&:: [Int] \to [Int]\\
&reverseAcc\ xs &&= rev'\ [\,]\ xs\\
&rev' &&:: [a] \to [a] \to [a]\\
&rev'\ as\ [\,] &&= as\\
&rev'\ as\ (x:xs) &&= rev'\ (x:as)\ xs
\end{aligned}
$$

Evaluating *reverseAcc* $[1, 2, 3]$, using both the call-by-value and call-by-name strategies gives the following evaluation trace:

	Expression	Space Profile
=	$\mathit{reverseAcc}\ (1 : (2 : (3 : [\,])))$	□□□□□□□□
=	$\mathit{rev}'\ [\,]\ (1 : (2 : (3 : [\,])))$	□□□□□□□□□
=	$\mathit{rev}'\ (1 : [\,])\ (2 : (3 : [\,]))$	□□□□□□□□□
=	$\mathit{rev}'\ (2 : (1 : [\,]))\ (3 : [\,])$	□□□□□□□□□
=	$\mathit{rev}'\ (3 : (2 : (1 : [\,])))\ [\,]$	□□□□□□□□□
=	$3 : (2 : (1 : [\,]))$	□□□□□□□
	Time steps = 5	Space units = 9

The above trace shows that the new definition for *reverse* gives improved performance, both in terms of the number of reduction steps and space units required. The improvements in time occur because applying the list constructor function : to the accumulator is more time-efficient than the append function ++ in the previous definition, which is recursive over its first argument. The space requirements are constant in this definition, in that only an additional amount of space is required to hold the accumulator. The reason for this is that as each element of the list to be reversed is encountered it is combined directly with the accumulator, so the expression does not expand as it is evaluated, as was the case in the original definition of *reverse*.

Proving time usage

In this section we have argued informally that the *sum* function takes $1+(2*n)$ steps, where n is the length of the input list. In order to formalise such a result, we write $\mid e \mid$ for the number of steps required to reduce the expression e. For simplicity, we will use call-by-value reduction and assume that all lists are finite, fully evaluated, and only contain integers.

Proposition 1 $\mid \mathit{sum}\ xs \mid = 1 + (2 * \mathit{length}\ xs)$

Proof by induction on xs:

Case : $[\,]$

$$
\begin{array}{ll}
 & |\ sum\ [\,]\ | \\
= & \quad \{\ \text{cost of applying the definition of } sum\ \} \\
 & 1 + |\ 0\ | \\
= & \quad \{\ 0 \text{ is a value}\ \} \\
 & 1 + 0 \\
= & \quad \{\ \text{arithmetic}\ \} \\
 & 1 + (2 * 0) \\
= & \quad \{\ \text{definition of } length\ \} \\
 & 1 + (2 * length\ [\,])
\end{array}
$$

Case : $(x : xs)$

$$
\begin{array}{ll}
 & |\ sum\ (x : xs)\ | \\
= & \quad \{\ \text{cost of applying the definition of } sum\ \} \\
 & 1 + |\ x + sum\ xs\ | \\
= & \quad \{\ \text{cost of applying addition}\ \} \\
 & 1 + |\ x\ | + |\ sum\ xs\ | + 1 \\
= & \quad \{\ \text{simplification}\ \} \\
 & 2 + |\ sum\ xs\ | \\
= & \quad \{\ \text{induction hypothesis}\ \} \\
 & 2 + 1 + (2 * length\ xs) \\
= & \quad \{\ \text{arithmetic}\ \} \\
 & 1 + (2 * (1 + length\ xs)) \\
= & \quad \{\ \text{definition of length}\ \} \\
 & 1 + (2 * (length\ (x : xs)))
\end{array}
$$

Note that this proof is not fully formal, in that neither the evaluation strategy nor the definition of $|\ e\ |$ have been formalised. One of the primary contributions of this thesis is to be able to make results like this precise.

Calculating time usage

Rather than simply proving the time complexity of *sum*, we can also systematically construct the time complexity, by starting from the assumption that it is linear, i.e. it can be expressed as some function f, initially unknown, on length of the list:

Proposition 2 $\mid sum\ xs \mid = f\ (length\ xs)$

We can now calculate f by induction on xs:
Case : $[\,]$

$$
\begin{array}{ll}
 & f\ (length\ [\,]) = \mid sum\ [\,] \mid \\
\Leftrightarrow & \quad \{\ \text{definition of } length\ \} \\
 & f\ 0 = \mid sum\ [\,] \mid \\
\Leftrightarrow & \quad \{\ \text{cost of applying the definition of } sum\ \} \\
 & f\ 0 = 1 + \mid 0 \mid \\
\Leftrightarrow & \quad \{\ 0 \text{ is a value}\ \} \\
 & f\ 0 = 1 + 0 \\
\Leftrightarrow & \quad \{\ \text{arithmetic}\ \} \\
 & f\ 0 = 1
\end{array}
$$

Case : $(x : xs)$

$$
\begin{array}{ll}
 & f\ (length\ (x : xs)) = \mid sum\ (x : xs) \mid \\
\Leftrightarrow & \quad \{\ \text{definition of } length\ \} \\
 & f\ (1 + length\ xs) = \mid sum\ (x : xs) \mid \\
\Leftrightarrow & \quad \{\ \text{cost of applying the definition of } sum\ \} \\
 & f\ (1 + length\ xs) = 1 + \mid x + sum\ xs \mid \\
\Leftrightarrow & \quad \{\ \text{cost of applying the definition of } +\text{, simplification}\ \} \\
 & f\ (1 + length\ xs) = 2 + \mid sum\ xs \mid \\
\Leftrightarrow & \quad \{\ \text{induction hypothesis}\ \} \\
 & f\ (1 + length\ xs) = 2 + f\ (length\ xs)
\end{array}
$$

$$
\begin{array}{ll}
\Leftarrow & \{ \text{ generalising } \mathit{length}\ \mathit{xs} \text{ to } n \ \} \\
& f\ (1+n) = 2 + f\ n \\
\Leftrightarrow & \{ \text{ arithmetic } \} \\
& f\ (n+1) = 2 + f\ n
\end{array}
$$

The above calculation shows that if $f\ 0 = 1$, then the base case of the inductive proof that $\mathit{sum}\ \mathit{xs} = f\ (\mathit{length}\ \mathit{xs})$ is satisfied, and that if $f\ (n+1) = 2 + f\ n$ then the inductive case is satisfied. Altogether, this gives us:

$$
\begin{array}{l}
|\ \mathit{sum}\ \mathit{xs}\ | = f\ (\mathit{length}\ \mathit{xs}) \\
\quad \textbf{where} \\
\quad\quad f\ 0 \qquad = 1 \\
\quad\quad f\ (n+1) = 2 + f\ n
\end{array}
$$

The recursive definition of f can be rewritten non-recursively as $f\ n = 1 + 2 * n$, which can be proved by simple induction over the natural number n.

1.5 Simplifying assumptions

Based upon the review in the previous section, for the remainder of this thesis we make a number of simplifying assumptions, which are summarised below:

- For the remainder of this thesis we only consider evaluation using strict, call-by-value, languages. In the call-by-name evaluation strategy, the time and space requirements are complicated by the fact that whether parts of a program are executed at all depends on their context within the program. For example, in this setting the requirements for evaluating an expression will only include the cost of evaluating those parts that are actually required in order to produce the result. By contrast, under call-by-value evaluation all parts of an expression will be evaluated, which admits reasoning in a compositional manner in the sense that the resource requirements of every part of the expression will contribute to the requirements of the whole expression.

- We choose to measure the time usage by counting evaluation steps, though at this point we don't specify what we consider as a step. In this way we can abstract away from real time costs, whilst still having an accurate model of time requirements, if an appropriate notion of evaluation step is chosen.
- The space usage will be taken as the maximum sized data structure produced during evaluation, in that we assume any space previously used, but not currently, may be re-used at a later time, i.e. that there is a garbage collector. Later on we will formalise what we mean by these data structures.

1.6 Related work

In this section we consider other work in the area of reasoning about the space and time requirements of functional programs. In order to relate the work to the main contributions of this thesis, we pay particular attention to what is considered a primitive unit of space or time, and how these units are then combined to express the overall requirements.

1.6.1 Time usage

First of all we look at reasoning about time usage and consider a number of different perspectives, in particular automated approaches to reasoning about time requirements, the challenges of lazy evaluation, and using types and monads to capture time information.

Automatic tools

The *Metric* system to automatically analyze the execution behaviour of first-order Lisp programs is presented by Wegbreit [5]. The system consists of three phases: local cost assignment, where a cost function is produced for the original program; recursion analysis, where the structure of the resulting local cost function is transformed to

yield a set of difference equations; and finally, difference equation solving to gain a closed-form solution.

In the initial local cost phase of the Metric system, a cost is assigned to primitive operations and non-recursive functions are given compositional costs generated from these. The costs of these primitive operations are stored in an external table which is then used to lookup a particular cost. In this way, the costs may either be constant, given by an expression, or be derived from experimental results on an actual machine. For example, an empty list may be given zero cost, a list $(x : xs)$ may be given a scalar expression of cost consisting of one (for the cons) plus the costs of x and xs, or a cost may be expressed as a performance tuple that captures the idea that an expression may not need to be fully evaluated to give a result.

Le Metayer's *ACE* (Automatic Complexity Evaluator) [6] system automatically analyses the execution time of programs in Backus' functional language FP, in order to give worst-case complexities. This system first derives a recursive complexity function, which is of the same structural form as the original function but instead counts the number of recursive calls, and then successively applies program transformations, from a built-in library of axioms, to get back a closed-form solution of the time usage.

In contrast to the previous system, ACE takes a macro-analysis view, where the time required to execute primitive operations is not considered. The aim here is to calculate a worst-case complexity asymptote, which is defined to be proportional to the number of non-primitive function calls.

Another framework for automatically calculating worst-case bounds for execution time is given by Rosendahl [7]. This system takes as input a first-order Lisp program and outputs a time bound function, based on call-by-value evaluation. The derivation of this function proceeds in two stages: the first is to produce a step-counting version of the input program, which takes the same arguments, but returns the computation time, and the second is to find a time bound function that is an upper-bound of the step-counting function. An optional third stage is to convert this time bound function into a closed-form. The step-counting function is generated from the original definition of the function by transforming it to return the number of sub-expressions

to be evaluated. For primitive functions, the step function counts one unit, and for non-primitives it counts one unit plus the time to evaluate the arguments and the function body.

Liu and Gomez [8] present a language-based approach to automatically generating worst-case time bounds for a first-order subset of Scheme, by applying techniques from static analysis and program transformation. As with previous work, the first step is to produce a timing function that takes the same arguments as the function under study and returns the execution time. The timing function is calculated compositionally (for call-by-value evaluation), but instead of assigning constant values to primitive functions, these are kept as symbolic constants and assigned concrete values generated by experimental measurements on an actual machine. This work is extended to consider higher-order functions in [9].

Automating complexity analysis is again addressed by Benzinger [10], to give upper bounds for first-order functional programs in NuPrl, a proof development system. The system uses symbolic evaluation of open terms to derive recursive cost equations, which are then solved by an external tool, in this case Mathematica. Costs are assigned to functions according to an operational call-by-name semantics, that counts the number of applications of the semantic rules until the expression is of normal form. This work is extended to higher-order functions in [11].

Lazy evaluation

The problem of calculating time bounds in the context of lazy evaluation is addressed by Wadler [12]. The notion of a projection transformer is used to express the context in which an expression is evaluated, in order to determine how much of the result is needed. A step-counting version of an expression is constructed by extending it with an extra argument to describe which part of the result is needed. The measurement of evaluation time is taken to be the number of steps required to reduce the expression to normal form, where a reduction is the application of a non-primitive function.

Complexity analysis for higher-order languages is considered by Sands [13], where

the cost of a function is expressed using a cost-closure to retain the evaluation history of higher-order functions, representing the cost of applying the function to its arguments. Following the use of projections by Wadler, this work was extended to consider lazy evaluation in [14].

Sands presents an alternative to cost-closures in [15], where instead a semantics based on bisimulation is used once the time cost has been exposed. This allows a form of equivalence to be defined in order to perform substitution of time rules. The unit of execution time is, once again, the number of non-primitive function calls.

A compositional approach to analysing the complexity of lazy first-order programs is presented by Bjerner and Holmström [16]. Instead of using projections to determine how much of the result is required, the approach is to use demand analysis to work backwards from the result. The measure of a time step in this case is to view each defining equation as a rewrite rule, and to count the number of applications of these that are required to produce a result, with primitive functions having no cost.

Types

Reistad and Gifford [17] take the approach of using types to capture execution costs. The motivation is that the execution costs will depend on the size of the inputs and these can be expressed in their types. The cost of functions, described as latent costs, are calculated by a syntax-driven semantics, which computes the cost of the sub-expressions and adds a symbolic constant representing the overhead of that construct. These constants are then determined experimentally on a physical machine and substituted into the overall cost analysis.

Similarly, Portillo & Hammond *et al* [18] introduce execution costs for strict evaluation of a higher-order language through sized types, and static analysis of costs is performed using type-inference mechanisms. This yields a cost expression that can be solved using an external constraint solver to give a closed form solution. Again, the resulting costs are parameterised by symbolic constants representing the cost of applying each rule in the language semantics, allowing only applications, for instance,

to be counted, or experimental values to be substituted instead. This work is extended to recursive languages in [19], though in this case beta-reductions are counted as primitive steps, with all other rules having zero application cost.

Abstract machines

Moran & Sands [20] model call-by-need evaluation using an operational semantics based on Sestoft's "mark 1" abstract machine. In earlier versions of this work, the execution time was taken to be the number of transitions of the abstract machine. However this is then simplified to only count the update transition, with all the other transitions assumed to be constant time and so not affecting the complexity of the overall computation. These costs are used in conjunction with a so-called tick algebra to represent and prove improvements and cost equivalences. A high-level semantics using evaluation contexts is introduced with a tick algebra, which is then used to make statements determining in what contexts the execution time of a computation may be improved.

Tick monad

A tick algebra approach has also been applied by Danielsson [21], to provide a semi-formal verification of time complexity for the dependently typed language Agda, under call-by-need evaluation. The approach is semi-formal in that it requires the user to annotate the program with ticks in appropriate locations. The recommended locations are at every rewriting of the left-hand side of a function definition by the right-hand side, i.e. for every function clause and lambda abstraction. The tick algebra is introduced as an annotated monad, with the time complexity then tracked using the type system.

Conclusion

Previous work in the area of time complexity for functional programs has often not paid much attention to the process of obtaining a step-counting version of the original

program. In particular, the unit of execution is typically just assumed to be the number of beta-reductions or non-primitive function calls. Attempts at making these measures more accurate, by producing an instrumented semantics with placeholders that are then filled with experimental values, gives results that are very dependent on a particular physical machine. A possible compromise between these two extremes would be the use of an appropriate abstract machine. Unlike the approach we shall develop, however, the existing work in this field doesn't provide an easy way of relating the low-level machine implementation to a high-level semantics.

1.6.2 Space

In contrast to the previous work on execution time, it is more common for analysis of space requirements to proceed using an abstract machine as a basis. In addition, term graphs and resource-aware operational semantics have also been used to model space requirements. We also consider research on space improvement in terms of the effect program transformation may have on space usage, how memory management may be improved for functional languages, and the notion of profiling.

Abstract machines

An abstract machine approach for describing the call-by-value evaluation of a lambda calculus variant is presented by Morrisett [22]. The purpose of the abstract machine is to model memory management, by exposing a heap, stack and environment, and modeling garbage collection. The work is extended in [23] to capture memory optimisations, such as efficient tail recursion and closure conversion. The motivation for this work is not to reason about programs, as it is a low-level semantics, but instead to prove properties about memory management strategies and garbage collection.

Hughes and Pareto [24] present a type and effect system for a strict first-order functional language, which guarantees that programs execute in constant space. The approach uses a modified SECD machine that makes the stack and heap space explicit, and maintains an upper bound on these storage constructs. The purpose of the system

is to verify upper bounds (by measuring the number of constructor symbols), but not to deduce these bounds themselves.

Term graphs

An accurate model for asymptotic space and time usage is presented by Bakewell and Runciman [25] for Core Haskell programs. The approach uses a term graph framework that makes sharing and addressing explicit, and provides a higher-level semantics than an abstract machine. Programs are represented as graphs that contain an expression to be evaluated, together with control and stack data structures. The space required to evaluate these graphs is defined to be the maximum number of nodes needed, and the time requirements are the number of steps to do so. The resulting framework is still low-level, and is not designed to be a standalone solution to performing space analysis of programs.

Operational semantics

An operational semantics that allows space profiling for the call-by-value lambda calculus is applied by Minamide [26] to consider the effects that program transformations have on space usage, in particular to show that applying the CPS transformation preserves space performance. This approach builds on profiling work for the parallel language NESL [27], presenting a big-step space semantics that is proved equivalent to an abstract machine implementation. However, Minamide notes that the treatment of space between both semantics is inconsistent, and shows how this can be corrected. One of our aims is to be able to move between high- and low-level semantics purely by calculation, and thus eliminate this type of problem.

Hofmann and Jost [28] present a model for statically predicting space usage, which considers how to obtain linear bounds on heap consumption for first-order functional programs. They define an operational semantics that maintains a count of the number of free memory cells, which is decremented when space is allocated and incremented

when space is deallocated. Using this approach, the space requirements for an expression are given by the number of initially free cells required to evaluate the expression. This semantics acts as the theory to a resource aware language, which can be used to give space bounds in terms of the size of the input. An extension to this technique has been developed to consider amortized costs for heap-space analysis [29].

Space improvement

Previous work on an improvement theory for time [20] is extended to consider space by Gustavsson and Sands [30], in order to to show when a program transformation will not lead to asymptotically worse space usage. As with the earlier work on time, Sestoft's abstract machine for the call-by-need lambda calculus is applied along with a tick algebra, which is modified to instead express heap and stack costs. In later work [31], the space model is then applied to consider the space behaviour of some common program transformations.

Hofmann [32] introduces a linear functional language that allows programs to be translated into C without any dynamic memory allocation. The system presents a safe way of performing in-place updates, instead of constructing the result of a function completely from new heap space. The approach proceeds by introducing a resource type which controls the number of constructor symbols and is used to ensure linear space requirements.

Profiling

As well as theoretical approaches to reasoning about space and time information, there are also practical approaches to obtaining this information, in the form of profilers.

The profiler distributed with the Glasgow Haskell Compiler gives feedback on both space and time usage [33]. Regular clock interrupts are used to measure how long a function has taken to complete, and the space requirements are given by measuring the size of all objects on the heap. The profiler differs to other approaches (such as Runciman and Wakeling's profiler [34]) in that the programmer can nominate *cost*

centres for costs to be attributed to, in order to avoid most costs going to commonly called functions, such as *map*. Later on in the thesis we will use this profiler for benchmarking the results that are produced by our theory.

Conclusion

Although previous work has considered the use of abstract machines to study space requirements, there has been no structured way of lifting this information to a high-level semantics. Our approach to addressing this problem is to present a systematic, purely calculational way of obtaining a high-level semantics that accurately reflects the behaviour of an underlying abstract machine.

1.7 Summary

This chapter provided an introduction to the analysis of space and time requirements for programs, and the specific issues that arise for functional languages. The simplifying assumptions we make for this thesis were explained, and finally, the chapter concluded with an overview of related work.

Chapter 2

Standard approach

In this chapter we set the scene for the rest of thesis by reviewing a traditional approach to reasoning about time usage using an instrumented semantics, but presented in a calculational style. In order to concentrate on the essence of the problem we first introduce a simpler language to study than Haskell. The properties required for this language are that it is sufficiently powerful, so that we can express some interesting programming examples, and also that it is small enough to facilitate proofs by hand. The lambda calculus [35] is an appropriate starting point because it has a compact syntax, consisting of only three constructs, and it is the *machine code* of functional programming, in that any expression in a high-level functional language can be translated to an equivalent expression in the lambda calculus. We begin with a brief review of the lambda calculus and its semantics, which is then used as a basis to illustrate a traditional approach to measuring time usage. This language is then extended with additional data structures and programming primitives to allow us to express more interesting programs, and calculate their time usage.

2.1 Lambda calculus

An expression in the lambda calculus is either a variable, an application of two expressions, or an abstraction, which consists of a variable and expression:

$$e ::= x \mid (e\ e) \mid (\lambda x.e)$$

To avoid over-bracketing of expressions, application is assumed to bracket to the left, while abstraction brackets to the right and is of lower precedence than application. As an example, the expression $\lambda x.\lambda y.f\,x\,y$ is read as $(\lambda x.(\lambda y.((f\,x)\,y)))$.

An occurrence of a variable may be categorized as either being *free* or *bound* in an expression, depending on its context. An abstraction $\lambda x.e$ binds the free occurrences of x in e, and is viewed as a function, with x as the argument and e as the function body. An expression that contains no free variables is described as *closed*, and these represent programs, because encountering unbound variables would indicate an error. Calculating the set of free and bound variables of an expression may be defined inductively over its structure as follows:

$$\begin{array}{lcllcl} \mathit{free}\ x & = & \{x\} & \mathit{bound}\ x & = & \{\} \\ \mathit{free}\ (f\,e) & = & \mathit{free}\ f \cup \mathit{free}\ e & \mathit{bound}\ (f\,e) & = & \mathit{bound}\ f \cup \mathit{bound}\ e \\ \mathit{free}\ (\lambda x.e) & = & \mathit{free}\ e - \{x\} & \mathit{bound}\ (\lambda x.e) & = & \mathit{bound}\ e \cup \{x\} \end{array}$$

Expressions are evaluated by performing substitutions on applications of abstractions: the expression $(\lambda x.e)\,e'$ is reduced by substituting all free occurrences of x in e by e', written as $e\,[e'/x]$. This is known as β-reduction or substitution and is once again

defined inductively over the structure of lambda expressions:

$$
\begin{array}{lcl}
y\,[e/x] & = & \begin{cases} e & \text{if } x = y \\ y & \text{otherwise} \end{cases} \\
(f\,e')\,[e/x] & = & (f\,[e/x])\;(e'\,[e/x]) \\
(\lambda y.e')\,[e/x] & = & \begin{cases} \lambda y.e' & \text{if } x = y \\ \lambda y.e'\,[e/x] & \text{if } x \neq y \wedge y \notin \mathit{free}\ e \\ \lambda z.e'\,[z/y]\,[e/x] & \text{where } z \notin \mathit{free}\ e \cup \mathit{free}\ e' \end{cases}
\end{array}
$$

Performing substitution is a simple idea but has complications, in that substituting an expression for a variable in another expression may cause name capture if a bound variable in the expression appears free in the substituted expression. For example, if the expression $(\lambda x.\lambda y.x\,y)\,y$ is β-reduced to produce $(\lambda y.x\,y)[y/x]$ without first renaming the variable y, then the result will be $\lambda y.y\,y$. To address this issue, the above definition of substitution first chooses a fresh variable name z, which doesn't occur free in either expression, and replaces all occurrences of y in the lambda expression with z, to ensure that there is no longer a name capture. In the case of our example this gives $(\lambda z.x\,z)[y/x]$, which can then be safely β-reduced to $\lambda z.y\,z$. In general, the process of renaming variables whilst preserving the structure of the expression is known as α-conversion.

2.2 Denotational semantics

Two popular forms of programming language semantics are denotational [36] and operational [37]. The purpose of a denotational semantics is to assign a value to every expression in the language. The semantics is defined using an environment ρ which, for the call-by-value evaluation strategy, maps variables to values. The concept of a value is made precise when we revisit this semantics in section 3.4. For

the lambda calculus, the denotation of an expression e, expressed as $[\![e]\!]$, is as follows:

$$\begin{array}{lcl} [\![x]\!] \ \rho & = & \rho\, x \\ [\![\lambda x.e]\!] \ \rho & = & \lambda v.[\![e]\!]\, \rho\{x \mapsto v\} \\ [\![f\, e]\!] \ \rho & = & ([\![f]\!]\, \rho)\, ([\![e]\!]\, \rho) \end{array}$$

That is, the meaning of a variable is the value that the variable maps to in the environment. In turn, an abstraction $\lambda x.e$ is interpreted by constructing a function that takes a value v and interprets e under the environment extended with x mapped to v. Finally, an application $f\, e$ is interpreted by applying the interpretation of f to the interpretation of e. In this way the semantics is described as *compositional*, in that the meaning of any construct in the language is defined purely in terms of the meaning of its parts.

Note that in order to define this semantics we need both syntactic and semantic views of abstraction and application, in that the lambda calculus syntax of abstraction is defined in terms of a built-in semantic abstraction, and similarly for application.

2.3 Operational semantics

In contrast to denotational semantics, an operational semantics defines *how* a program is executed, at a suitable level of abstraction. There are two commonly used forms of operational semantics, which differ in the size of step, or *granularity*, that is counted as atomic. A fine-grained approach is given in the form of a small-step semantics [37]. This provides a low-level view, where each individual evaluation step is visible. In contrast, a coarser approach in given in a big-step semantics [38]. This semantics provides a more abstract concept of evaluation, in that it can be thought to occur in one big step with the precise details hidden.

As a starting point to statically reason about time usage, using a denotational approach to semantics does not seem an appropriate choice, because it is concerned with what function a program computes, without worrying about how it is carried out. For this reason, we first consider an operational semantics. A small-step semantics

will be provided as the specification for the language, as the concept of counting evaluation steps is more apparent at this level. A big-step semantics will then be introduced for reasoning about the language, because it provides a more high-level view. These two semantics will then be shown to be equivalent.

2.4 Small-step semantics

A small-step reduction relation $\rightarrow$ on expressions can be inductively defined for call-by-value evaluation using the following three rules:

$$\frac{}{(\lambda x.e)\, v \rightarrow e[v/x]}\ \beta_{\rightarrow}$$

$$\frac{f \rightarrow f'}{f\, e \rightarrow f'\, e}\ \text{FUN}_{\rightarrow} \qquad \frac{e \rightarrow e'}{v\, e \rightarrow v\, e'}\ \text{FUN2}_{\rightarrow}$$

The rule $\beta_{\rightarrow}$ states that an application $(\lambda x.e)\, v$, where the left-hand side is an abstraction and the right-hand side is a value, can be reduced by performing β-reduction and replacing all free occurrences of x in e by v, written as $e[v/x]$. The FUN$_{\rightarrow}$ rule states that an application may make a transition if the left-hand side can make a transition. In turn, when the left-hand side of an application is a value, then the right-hand side may make a transition (FUN2$_{\rightarrow}$ rule). For the pure lambda calculus abstractions are the only kind of values, as will be shown in section 2.5.

The $\beta_{\rightarrow}$ rule is the key rule in this semantics, in that this is where the computation occurs. The other two rules are just structural, in that they determine where the $\beta_{\rightarrow}$rule can be applied. This idea, of locating where computation can occur, will be revisited when we consider evaluation contexts in section 2.9.

The small-step rules each describe a single reduction step. Complete evaluation, from an expression to a value, can be performed by successively applying these rules until the expression can no longer be further reduced. This is expressed by the relation $\overset{*}{\rightarrow}$, defined as the reflexive/transitive closure of the relation $\rightarrow$. This relation can be

extended to incorporate counting the number of small-step rules applied. We define a step counting relation $\xrightarrow{n}$ using the following rules:

$$\frac{}{e \xrightarrow{0} e} \qquad \frac{\exists e'.e \to e' \wedge e' \xrightarrow{n} e''}{e \xrightarrow{n+1} e''}$$

That is, an expression reduces to itself in zero steps, and, if there exists an expression e' such that $e \to e'$, then the number of steps to reduce e is one plus the number to reduce e'.

2.5 Rule induction

In order to prove properties of evaluation we will use the principle of rule induction, which will be introduced with a simple example. Suppose that a set X is inductively defined by the following two rules:

$$\frac{}{a \in X} \qquad \frac{x \in X}{f(x) \in X}$$

These rules state that the constant a is in the set X and, if x is in X, then the function f applied to x is also an element of X. Moreover, the inductive nature of the rules means that these are the only elements of X, or more formally, that X is the smallest set defined by the two rules. In order to prove that a property P holds for all x in X, rule induction states that it is sufficient to show that P holds for a and that, assuming P holds for x, it also holds for $P(f(x))$. That is:

$$\frac{\begin{array}{l} P(a) \\ \forall x \in X.(P(x) \Rightarrow P(f(x))) \end{array}}{\forall x \in X.P(x)}$$

As an example, the set of natural numbers can be expressed in the above form, by

taking $\mathbb{N}$ for X, *Zero* for a and *Succ* for f, as shown below:

$$\frac{}{Zero \in \mathbb{N}} \qquad \frac{n \in \mathbb{N}}{Succ\ n \in \mathbb{N}}$$

Then, in order to prove a property P holds for all natural numbers, it is sufficient to show that P holds for *Zero* and, assuming P holds for n, show that it holds for $P(Succ\ n)$, which is just the familiar principle of mathematical induction:

$$\frac{\begin{array}{l} P(Zero) \\ \forall n \in \mathbb{N}.P(n) \Rightarrow P(Succ\ n) \end{array}}{\forall n \in \mathbb{N}.P(n)}$$

Rule induction for $\rightarrow$ and $\overset{n}{\rightarrow}$

To prove that a property $P(e, e')$ holds for all transitions $e \rightarrow e'$ in the small-step semantics for the lambda calculus, we consider the three cases of each transition and appeal to rule induction to obtain the following induction principle:

$$\frac{\begin{array}{l} P((\lambda x.e)\, v, e[v/x]) \\ \forall f \rightarrow f'.P(f, f') \Rightarrow P(f\, e, f'\, e) \\ \forall e \rightarrow e'.P(e, e') \Rightarrow P(v\, e, v\, e') \end{array}}{\forall e \rightarrow e'.P(e, e')}$$

In the same way, to prove that P holds for all $e \overset{n}{\rightarrow} e''$, we apply rule induction to the two inference rules that define $e \overset{n}{\rightarrow} e''$, to give the following induction principle:

$$\frac{\begin{array}{l} P(e, e, 0) \\ \forall e, e', e'', n.e \rightarrow e' \wedge P(e', e'', n) \Rightarrow P(e, e'', n+1) \end{array}}{\forall e \overset{n}{\rightarrow} e'.P(e, e', n)}$$

These induction principles will now be used in the following proofs.

Determinacy

The small-step transition function is deterministic, meaning that for any expression there is never a choice of redex to reduce.

Proposition 3 $\forall e, e', e''.\, e \to e' \wedge e \to e'' \Rightarrow e' \equiv e''$

This can be proved by rule induction on $e \to e'$:

$$
\begin{array}{ll}
 & \forall e, e', e''.e \to e' \wedge e \to e'' \Rightarrow e' \equiv e'' \\
= & \quad \{ \text{ currying } \} \\
 & \forall e, e', e''.e \to e' \Rightarrow (e \to e'' \Rightarrow e' \equiv e'') \\
= & \quad \{ \text{ logic, lemma 1 } \} \\
 & \forall e, e'.e \to e' \Rightarrow (\forall e''.e \to e'' \Rightarrow e' \equiv e'') \\
= & \quad \{ \text{ define } P(e, e') = \forall e''.e \to e'' \Rightarrow e' \equiv e'' \} \\
 & \forall e, e'.e \to e' \Rightarrow P(e, e') \\
\Leftarrow & \quad \{ \text{ rule induction } \} \\
 & P((\lambda x.e)v, e[v/x]) \\
 & \forall f, f'.f \to f' \Rightarrow P(f, f') \Rightarrow P(f\, e, f'\, e) \\
 & \forall e, e'.e \to e' \Rightarrow P(e, e') \Rightarrow P(v\, e, v\, e') \\
= & \quad \{ \text{ definition of } P \} \\
 & (1)\ \forall e''.(\lambda x.e)v \to e'' \Rightarrow e'' \equiv e[v/x] \\
 & (2)\ \forall f, f'.f \to f' \Rightarrow (\forall f''.f \to f'' \Rightarrow f'' \equiv f') \Rightarrow (\forall e'.f\, e \to e' \Rightarrow e' \equiv f'\, e) \\
 & (3)\ \forall e, e'.e \to e' \Rightarrow (\forall e''.e \to e'' \Rightarrow e'' \equiv e') \Rightarrow (\forall v'.v\, e \to v' \Rightarrow v' \equiv v\, e')
\end{array}
$$

Proof of (1) :

$$
\begin{array}{ll}
 & (\lambda x.e)v \to e'' \Rightarrow e'' \equiv e[v/x] \\
= & \quad \{ \text{ rule inspection } \} \\
 & \text{true}
\end{array}
$$

The $\beta_{\to}$ rule is the only one that can be applied in this example. In particular, the premise of the FUN$_{\to}$ rule is not satisfied because the left hand side can not make

a transition, and the $\text{FUN}_{2\rightarrow}$ rule can not be applied because the right side of the application is a value and therefore can not make a transition.

Proof of (2) :

$$
\begin{aligned}
& \forall f, f'.f \rightarrow f' \Rightarrow (\forall f''.f \rightarrow f'' \Rightarrow f'' \equiv f') \Rightarrow (\forall e'.f\, e \rightarrow e' \Rightarrow e' \equiv f'\, e) \\
= & \quad \{ \text{ currying } \} \\
& \forall f, f'.f \rightarrow f' \wedge (\forall f''.f \rightarrow f'' \Rightarrow f'' \equiv f') \Rightarrow (\forall e'.f\, e \rightarrow e' \Rightarrow e' \equiv f'\, e) \\
= & \quad \{ \text{ logic } \} \\
& \forall f, f', e'.f \rightarrow f' \wedge (\forall f''.f \rightarrow f'' \Rightarrow f'' \equiv f') \Rightarrow f\, e \rightarrow e' \Rightarrow e' \equiv f'\, e \\
= & \quad \{ \text{ currying } \} \\
& \forall f, f', e'.f \rightarrow f' \wedge (\forall f''.f \rightarrow f'' \Rightarrow f'' \equiv f') \wedge f\, e \rightarrow e' \Rightarrow e' \equiv f'\, e \\
= & \quad \{ \text{ rule inspection } \} \\
& \text{true}
\end{aligned}
$$

The $\text{FUN}_{\rightarrow}$ rule is the only one that can be applied because f can make a transition to f' so it must not be an abstraction, and therefore neither the $\beta_{\rightarrow}$ or $\text{FUN}_{2\rightarrow}$ rule can be applied.

Proof of (3) :

$$
\begin{aligned}
& \forall e, e'.e \rightarrow e' \Rightarrow (\forall e''.e \rightarrow e'' \Rightarrow e'' \equiv e') \Rightarrow (\forall v'.v\, e \rightarrow v' \Rightarrow v' \equiv v\, e') \\
= & \quad \{ \text{ currying, logic } \} \\
& \forall e, e', v'.e \rightarrow e' \wedge (\forall e''.e \rightarrow e'' \Rightarrow e'' \equiv e') \wedge v\, e \rightarrow v' \Rightarrow v' \equiv v\, e' \\
= & \quad \{ \text{ rule inspection } \} \\
& \text{true}
\end{aligned}
$$

The $\text{FUN}_{2\rightarrow}$ rule is the only one that can be applied because v cannot make a transition and hence the premise of the $\text{FUN}_{\rightarrow}$ rule is not satisfied. Moreover, e can make a transition so it is not a value, and therefore the $\beta_{\rightarrow}$ rule can not be applied.

Lemma 1 $(\forall x.A \Rightarrow P(x)) = (A \Rightarrow \forall x.P(x))$

The auxiliary logical result used in the proof above is verified, using natural deduction, as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{[\forall x.A \Rightarrow P(x)]}{A \Rightarrow P(z)}\forall - E \quad [A]}{P(z)}\Rightarrow\text{-E}}{\forall x.P(x)}\forall - I}{A \Rightarrow \forall x.P(x)}\Rightarrow -I}{(\forall x.A \Rightarrow P(x)) \Rightarrow (A \Rightarrow \forall x.P(x))}\Rightarrow\text{-I} \qquad \dfrac{\dfrac{\dfrac{\dfrac{[A \Rightarrow \forall x.P(x)] \quad [A]}{\forall x.P(x)}\forall - I}{P(z)}\forall - E}{A \Rightarrow P(z)}\Rightarrow\text{-I}}{\forall x.A \Rightarrow P(x)}\forall - I}{(A \Rightarrow \forall x.P(x)) \Rightarrow (\forall x.A \Rightarrow P(x))}\Rightarrow\text{-I}$$

$$\dfrac{(\forall x.A \Rightarrow P(x)) \Rightarrow (A \Rightarrow \forall x.P(x)) \qquad (A \Rightarrow \forall x.P(x)) \Rightarrow (\forall x.A \Rightarrow P(x))}{(\forall x.A \Rightarrow P(x)) = (A \Rightarrow \forall x.P(x))}\wedge\text{-I}$$

Values

A closed expression e is a value if and only if e is an abstraction, i.e. it is of the form $\lambda x.e'$. This equivalence can be shown by proving both sides of the double implication $e \not\rightarrow \;\Leftrightarrow\; e = \lambda x.e'$, where $e \not\rightarrow$ iff $\neg\exists e'.e \rightarrow e'$.

Lemma 2 (Every abstraction is a value) *For all closed expressions of the form $\lambda x.e$, we have $\lambda x.e \not\rightarrow$*

By rule inspection $\lambda x.e$ matches no left-hand side of the rules and so cannot make a transition.

Lemma 3 (If a closed expression is a value then it is an abstraction) *For all closed expressions e, if $e \not\rightarrow$ then $e \equiv \lambda x.e'$*

This is achieved by proving the contrapositive statement, namely that if e is not an abstraction then it is not a value. By structural induction on e, if e is not an abstraction then it must be an application (for closed expressions). Case analysis can then be performed on the structure of the application, where both sides of the application may either be an abstraction, or not an abstraction.

Case : $e = (\lambda x.e)\,(\lambda y.e')$

$$
\begin{array}{ll}
 & (\lambda x.e)\,(\lambda y.e') \\
\rightarrow & \quad \{\ \beta_{\rightarrow} \text{ rule } \} \\
 & e\,[(\lambda y.e')/x]
\end{array}
$$

Case : $e = (\lambda x.e)\,e'$ where e' is not an abstraction

$$
\begin{array}{ll}
 & (\lambda x.e)\,e' \\
\Rightarrow & \quad \{\ \text{induction hypothesis, } e' \text{ is not an abstraction } \} \\
 & \exists e''.(\lambda x.e)\,e' \wedge e' \rightarrow e'' \\
\rightarrow & \quad \{\ \text{FUN}_{2\rightarrow} \} \\
 & (\lambda x.e)\,e''
\end{array}
$$

Case : $e = f\,e$ where f is not an abstraction

$$
\begin{array}{ll}
 & f\,e \\
\Rightarrow & \quad \{\ \text{induction hypothesis, } f \text{ is not an abstraction } \} \\
 & \exists f'.f\,e \wedge f \rightarrow f' \\
\rightarrow & \quad \{\ \text{FUN}_{\rightarrow} \} \\
 & f'\,e
\end{array}
$$

Therefore, all possible applications can make a transition so they are not values.

2.6 Big-step semantics

An operational semantics for the call-by-value lambda calculus can also be defined using a big-step semantics. In particular, we write $e \Downarrow v$ to mean that the expression e can be evaluated to the value v, where the big-step relation $\Downarrow$ is defined by the following two rules:

$$\frac{}{v \Downarrow v}\ \text{VAL}_{\Downarrow} \qquad \frac{f \Downarrow \lambda x.e' \qquad e \Downarrow v' \qquad e'\,[v'/x] \Downarrow v}{f\,e \Downarrow v}\ \text{APP}_{\Downarrow}$$

The rule $\text{VAL}_{\Downarrow}$ states that values cannot be further evaluated. In turn, the $\text{APP}_{\Downarrow}$ rule states that an application $f\,e$ can be reduced to a value v if f can be reduced to an abstraction $\lambda x.e'$ and e to a value v', and then e' with v' β-substituted for x reduced to v. Unlike in the small-step semantics, the big-step rules do not explicitly define the order of reduction, though the β-substitution step requires the result of reducing both f and e, so these would have to occur first. In this way, the big-step semantics provides greater abstraction from the details of evaluation.

Evaluating an expression using a big-step semantics is performed by building a derivation tree, as shown in the following example:

$$\dfrac{\dfrac{}{\lambda x.x \Downarrow \lambda x.x}\,\text{VAL}_{\Downarrow} \qquad \dfrac{}{\lambda y.y\,y \Downarrow \lambda y.y\,y}\,\text{VAL} \qquad \dfrac{}{(\lambda x.x)\,[(\lambda y.y\,y)/x] \Downarrow \lambda y.y\,y}\,\text{VAL}_{\Downarrow}}{(\lambda x.x)\,(\lambda y.y\,y) \Downarrow \lambda y.y\,y}\,\text{APP}_{\Downarrow}$$

The expression to the left of the $\Downarrow$ relation is expanded until the axiom $\text{VAL}_{\Downarrow}$ can be applied. The resulting value is then passed back down the tree.

Step counting big-step semantics

The big-step rules may also be instrumented to count the number of steps. In this case the number of applications of the $\text{APP}_{\Downarrow}$ rule is counted, as follows:

$$\dfrac{}{v \Downarrow^0 v}\,\text{VAL}_{\Downarrow} \qquad \dfrac{f \Downarrow^a \lambda x.e' \qquad e \Downarrow^b v' \qquad e'[v'/x] \Downarrow^c v}{fe \Downarrow^{a+b+c+1} v}\,\text{APP}_{\Downarrow}$$

These rules state that the number of steps to evaluate a value is zero, and the number of steps to evaluate an application is one plus the number to evaluate each of its constituent parts. Counting the number of applications can be shown to be equivalent to counting the number of small-step rules by showing that this new step counting semantics is equivalent to the instrumented small-step semantics defined earlier.

2.7 Soundness and completeness

The small-step and big-step operational semantics presented in the previous sections can be shown to be equivalent in that if an expression e reduces to v in n small-steps, then it will take the same number of steps to be evaluated using the big-step semantics, and vice versa. This equivalence can be formalised as follows:

Proposition 4 $e \xrightarrow{n} v \;\Leftrightarrow\; e \Downarrow^n v$

Again, the equivalence can be proved by showing that an implication holds in both directions. We want to prove the correctness of the big-step counting semantics, with respect to the small-step semantics, so the backward direction can be understood as *soundness*, expressing that every element in the relation $\xrightarrow{n}$ is also in $\Downarrow^n$. Conversely, the forward direction corresponds to *completeness*, which expresses that there are no more elements in the relation $\Downarrow^n$ than there are in $\xrightarrow{n}$.

Lemma 4 (Soundness) $e \Downarrow^n v \Rightarrow\; e \xrightarrow{n} v$

Proof by rule induction over $e \Downarrow^n v$:

Case : $v \Downarrow^0 v$
That $\lambda x.e \xrightarrow{0} \lambda x.e$ is trivial from the definition of $\xrightarrow{n}$.

Case : $f\, e \Downarrow^{n+m+1} v$
Assuming $f \Downarrow^n \lambda x.e', (\lambda x.e')\, e \Downarrow^m v, f \xrightarrow{n} \lambda x.e', (\lambda x.e')\, e \xrightarrow{m} v$, show that $f\, e \xrightarrow{n+m+1} v$:

$$
\begin{array}{ll}
 & f\, e \\
\xrightarrow{n} & \quad \{\ \text{inductive assumption}\ \} \\
 & (\lambda x.e')\, e \\
\xrightarrow{1} & \quad \{\ \beta_{\rightarrow}\ \text{rule}\ \} \\
 & e'\,[e/x] \\
\xrightarrow{m} & \quad \{\ \text{inductive assumption}\ \} \\
 & v
\end{array}
$$

Then the required result, $f\, e \xrightarrow{n+m+1} v$, follows by transitivity (lemma 5).

Transitivity of $\stackrel{n}{\rightarrow}$ relation

Lemma 5 $\forall x, y, z. x \stackrel{a}{\rightarrow} y \wedge y \stackrel{b}{\rightarrow} z \Rightarrow x \stackrel{a+b}{\rightarrow} z$

Proof by rule induction over $x \stackrel{a}{\rightarrow} y$:

$$
\begin{array}{ll}
 & \forall x, y, z.\ x \stackrel{a}{\rightarrow} y \wedge y \stackrel{b}{\rightarrow} z \Rightarrow x \stackrel{a+b}{\rightarrow} z \\
= & \quad \{\ \text{currying}\ \} \\
 & \forall x, y, z.\ x \stackrel{a}{\rightarrow} y \Rightarrow y \stackrel{b}{\rightarrow} z \Rightarrow x \stackrel{a+b}{\rightarrow} z \\
= & \quad \{\ \text{logic}\ \} \\
 & \forall x, y.\ x \stackrel{a}{\rightarrow} y \Rightarrow \forall z. y \stackrel{b}{\rightarrow} z \Rightarrow x \stackrel{a+b}{\rightarrow} z \\
= & \quad \{\ \text{define } P(x, y, a) = \forall z, b. y \stackrel{b}{\rightarrow} z \Rightarrow x \stackrel{a+b}{\rightarrow} z\ \} \\
 & \forall x, y.\ x \stackrel{a}{\rightarrow} y \Rightarrow P(x, y, a) \\
\Leftarrow & \quad \{\ \text{rule induction}\ \} \\
 & P(e, e, 0) \\
 & \forall e, e', e'', n.\ e \rightarrow e' \wedge P(e', e'', n) \Rightarrow P(e, e'', n+1) \\
= & \quad \{\ \text{definition of } P\ \} \\
 & (1)\ \ \forall z, b.\ e \stackrel{b}{\rightarrow} z \Rightarrow e \stackrel{0+b}{\rightarrow} z \\
 & (2)\ \ \forall e, e', e'', n.\ e \rightarrow e' \wedge (\forall z, b. e'' \stackrel{b}{\rightarrow} z \Rightarrow e' \stackrel{n+b}{\rightarrow} z) \Rightarrow (\forall z', b'. e'' \stackrel{b'}{\rightarrow} z' \Rightarrow e \stackrel{n+1+b'}{\rightarrow} z')
\end{array}
$$

Proof of (1) :

$$
\begin{array}{ll}
 & e \stackrel{b}{\rightarrow} z \\
= & \quad \{\ \text{arithmetic}\ \} \\
 & e \stackrel{0+b}{\rightarrow} z
\end{array}
$$

Proof of (2) :

$$
\begin{array}{ll}
 & \forall e, e', e'', n.\ e \rightarrow e' \wedge (\forall z, b. e'' \stackrel{b}{\rightarrow} z \Rightarrow e' \stackrel{n+b}{\rightarrow} z) \Rightarrow (\forall z', b'. e'' \stackrel{b'}{\rightarrow} z' \Rightarrow e \stackrel{n+1+b'}{\rightarrow} z') \\
= & \quad \{\ \text{logic}\ \} \\
 & \forall e, e', e'', n, z', b'.\ e \rightarrow e' \wedge (\forall z, b. e'' \stackrel{b}{\rightarrow} z \Rightarrow e' \stackrel{n+b}{\rightarrow} z) \Rightarrow e'' \stackrel{b'}{\rightarrow} z' \Rightarrow e \stackrel{n+1+b'}{\rightarrow} z' \\
= & \quad \{\ \text{currying}\ \}
\end{array}
$$

$$\begin{array}{ll} & \forall e, e', e'', n, z', b'.\ e \to e' \wedge (\forall z, b. e'' \xrightarrow{b} z \Rightarrow e' \xrightarrow{n+b} z) \wedge e'' \xrightarrow{b'} z' \Rightarrow e \xrightarrow{n+1+b'} z' \\ = & \quad \{\ \forall\text{-elimination}\ \} \\ & \forall e, e', e'', n, z', b'.\ e \to e' \wedge e' \xrightarrow{n+b'} z' \Rightarrow e \xrightarrow{n+1+b'} z' \\ = & \quad \{\ \text{definition of } \xrightarrow{n}\ \} \\ & \forall e, e', e'', n, z', b'.\ e \xrightarrow{n+1+b'} z' \Rightarrow e \xrightarrow{n+1+b'} z' \\ = & \quad \{\ \text{logic}\ \} \\ & \text{true} \end{array}$$

Lemma 6 (Completeness) $e \xrightarrow{n} v \ \Rightarrow\ e \Downarrow^n v$

Proof by rule induction over $e \xrightarrow{n} v$:

Case : $v \xrightarrow{0} v$

Show that $v \Downarrow^0 v$. This is trivially true by the $\text{VAL}_\Downarrow$ rule.

Case : $e \xrightarrow{n+1} v$

Assuming $\forall e, e', e''. e \to e' \wedge e' \xrightarrow{n} v \Rightarrow e' \Downarrow^n v$, show that $e \Downarrow^{n+1} v$:

$$\begin{array}{ll} & e \to e' \wedge e' \Downarrow^n v \ \Rightarrow\ e \Downarrow^{n+1} v \\ = & \quad \{\ \text{logic}\ \} \\ & e \to e' \ \Rightarrow\ e' \Downarrow^n v \ \Rightarrow\ e \Downarrow^{n+1} v \\ = & \quad \{\ \text{lemma 7}\ \} \\ & \text{true} \end{array}$$

Lemma 7 $e \to e' \ \Rightarrow\ e' \Downarrow^n v \ \Rightarrow\ e \Downarrow^{n+1} v$

Proof by rule induction on $e \to e'$:

Case : $(\lambda x.e)\, v' \to e\,[v'/x]$

Show that $e\,[v'/x] \Downarrow^n v \ \Rightarrow\ (\lambda x.e)\, v' \Downarrow^{n+1} v$

$$\begin{array}{ll} & e\,[v'/x] \Downarrow^n v \\ = & \quad \{\ \wedge\text{-introduction, } \text{VAL}_\Downarrow \text{ rule}\ \} \end{array}$$

$$
\begin{array}{ll}
& v' \Downarrow^0 v' \ \wedge\ \lambda x.e \Downarrow^0 \lambda x.e \ \wedge\ e\,[v'/x] \Downarrow^n v \\
\Rightarrow & \quad \{\ \text{APP}_{\Downarrow}\ \text{rule}\ \} \\
& (\lambda x.e)\, v' \Downarrow^{0+0+n+1} v \\
= & \quad \{\ \text{arithmetic}\ \} \\
& (\lambda x.e)\, v' \Downarrow^{n+1} v
\end{array}
$$

Case : $f\,e \to f'\,e$

Assuming $f \to f', f' \Downarrow^n v \Rightarrow f \Downarrow^{n+1} v$, show that $f'\,e \Downarrow^m v \Rightarrow f\,e \Downarrow^{m+1} v$

$$
\begin{array}{ll}
& f'\,e \Downarrow^m v \\
\Rightarrow & \quad \{\ \text{APP}_{\Downarrow}\ \text{rule (backwards)}\ \} \\
& \exists a, b, c, x, e', v'.\ e \Downarrow^a v' \ \wedge\ f' \Downarrow^b \lambda x.e' \ \wedge\ e'\,[v'/x] \Downarrow^c v \ \wedge\ m = a+b+c+1 \\
\Rightarrow & \quad \{\ \text{inductive assumption}\ \} \\
& \exists a, b, c, x, e', v'.\ e \Downarrow^a v' \ \wedge\ f \Downarrow^{b+1} \lambda x.e' \ \wedge\ e'\,[v'/x] \Downarrow^c v \ \wedge\ m = a+b+c+1 \\
\Rightarrow & \quad \{\ \text{APP}_{\Downarrow}\ \text{rule},\ a+1 = m-b-c\ \} \\
& f\,e \Downarrow^{m-b-c+b+c+1} v \\
\Rightarrow & \quad \{\ \text{arithmetic}\ \} \\
& f\,e \Downarrow^{m+1} v
\end{array}
$$

Case : $ve \to v\,e'$ Assuming $e \to e', e' \Downarrow^n v \Rightarrow e \Downarrow^{n+1} v$, show that $f\,e' \Downarrow^m v \Rightarrow f\,e \Downarrow^{m+1} v$:

$$
\begin{array}{ll}
& f\,e' \Downarrow^m v \\
\Rightarrow & \quad \{\ \text{APP}_{\Downarrow}\ \text{rule (backwards)}\ \} \\
& \exists a, b, c, x, e'', v'.\ e \Downarrow^a v' \ \wedge\ f \Downarrow^b \lambda x.e'' \ \wedge\ e'\,[v'/x] \Downarrow^c v \ \wedge\ m = a+b+c+1 \\
\Rightarrow & \quad \{\ \text{inductive assumption}\ \} \\
& \exists a, b, c, x, e'', v'.\ e \Downarrow^{a+1} v' \ \wedge\ f \Downarrow^b \lambda x.e'' \ \wedge\ e'\,[v'/x] \Downarrow^c v \ \wedge\ m = a+b+c+1 \\
\Rightarrow & \quad \{\ \text{APP}_{\Downarrow}\ \text{rule},\ a+1 = m-b-c\ \} \\
& f\,e \Downarrow^{m-b-c+b+c+1} v \\
\Rightarrow & \quad \{\ \text{arithmetic}\ \} \\
& f\,e \Downarrow^{m+1} v
\end{array}
$$

2.8 Extended language

The lambda calculus can express any computable function by using suitable encodings, such as Church numerals [35]. However, these low-level encodings would mask the space and time behaviour by introducing artificial overheads. For example, evaluating the addition of integers encoded as Church numerals would take time proportional to the first integer. The lambda calculus can be extended with data structures, primitive functions and control structures in order to give a more realistic model.

Extended language syntax

For our purposes we extend the lambda calculus with natural numbers, booleans and lists, together with a number of primitive functions on these types, as well as conditional expressions and recursion, by means of a fixpoint operator μ:

$$\begin{aligned} e ::= \;& x \mid \lambda x.e \mid e\,e \\ \mid \;& \mathbb{N} \mid \mathbb{B} \mid [\,] \mid e : e \\ \mid \;& \mathit{add}\ e\ e \mid \mathit{null}\ e \mid \mathit{head}\ e \mid \mathit{tail}\ e \\ \mid \;& \textbf{if}\ e\ \textbf{then}\ e\ \textbf{else}\ e \mid \mu x.e \end{aligned}$$

As an example, the function that appends two lists, expressed in Haskell as

$$\mathit{append}\ \mathit{xs}\ \mathit{ys} = \textbf{if}\ (\mathit{null}\ \mathit{xs})\ \textbf{then}\ [\,]\ \textbf{else}\ (\mathit{head}\ \mathit{xs}) : (\mathit{append}\ (\mathit{tail}\ \mathit{xs})\ \mathit{ys})$$

may be expressed similarly in the language as follows:

$$\mu\ \mathit{append}.(\lambda \mathit{xs}.\lambda \mathit{ys}.\textbf{if}\ (\mathit{null}\ \mathit{xs})\ \textbf{then}\ [\,]\ \textbf{else}\ (\mathit{head}\ \mathit{xs}) : (\mathit{append}\ (\mathit{tail}\ \mathit{xs})\ \mathit{ys}))$$

The primitive functions are implemented as fully saturated, meaning that they take all their arguments at once and so cannot be partially applied. The reason for this is to make it easier to define what a value is: if primitives were introduced as constants then, for example, *add* 1 would be a value, because it cannot be further evaluated. This does not affect what can be expressed in the language, because partial application can still be implemented using abstraction. For example, the function *add* 1 can be

re-expressed as $(\lambda x.add\ x\ 1)$. We also extend the notion of values to include natural numbers, booleans and lists containing further values:

$$v\ ::=\ \lambda x.e \mid \mathbb{N} \mid \mathbb{B} \mid [\,] \mid v : v$$

The language is presented as untyped because we are only interested in analysing the space and time properties of well-formed expressions. Adding a type system would be straightforward, but is not our concern here.

2.9 Evaluation contexts

Evaluation of the new structures in the language could be defined by extending the small-step rules, but because extra rules are required to show how the arguments to functions are evaluated if the function can not yet be applied, these can quickly become unwieldy. Instead, the new small-step semantics can be expressed more succinctly using the notion of evaluation contexts [39], which specify the position at which reduction can take place within an expression, together with a set of reduction rules which define how reduction may occur in these positions.

Contexts for the extended lambda calculus

The structure of evaluation contexts of the language is similar to that of the language itself, but has a unique occurrence of a *hole* as a sub-expression, written as $[-]$, which represents the point at which reduction can occur:

$$\begin{aligned} C ::=\ & [-] \mid v\,C \mid C\,e \\ \mid\ & v : C \mid C : e \\ \mid\ & add\ v\ C \mid add\ C\ e \mid null\ C \mid head\ C \mid tail\ C \\ \mid\ & \textbf{if}\ C\ \textbf{then}\ e\ \textbf{else}\ e \end{aligned}$$

The grammar describes the positions at which reduction can occur within an expression under call-by-value evaluation. If the expression is an application where the

left-hand side is a value, then the reduction hole is located in the right-hand expression, otherwise it is in the left-hand expression. Similarly, in the list constructor (:) and *add* case the evaluation order is forced to be left-to-right by only looking for the hole in the right expression when the left one is a value. The location of the hole in the *null*, *head* and *tail* functions is trivial because they have only one argument, and, if the expression is a conditional, then the position of the hole is in the *if*-part.

Once the hole has been located in an expression it can be filled by applying the appropriate small-step rule to the subexpression replaced by the hole, and then plugging the result back into the hole. This is expressed using the following rule:

$$\frac{e \to e'}{C[e] \to C[e']}\ \text{CON}_{\to}$$

The application of a context C to an expression e, expressed as $C[e]$, is defined by straightforward recursion over the structure of C. The small-step semantics for the extended language can then be defined using the following rules, where n and m represent natural numbers and v and vs are values:

$(\lambda x.e)\, v \to e\,[v/x]$	$\beta_{\to}$
$add\ n\ m \to n + m$	$\text{ADD}_{\to}$
$null\ [\,] \to true$	$\text{NULL1}_{\to}$
$null\ (v : vs) \to false$	$\text{NULL2}_{\to}$
$head\ (v : vs) \to v$	$\text{HEAD}_{\to}$
$tail\ (v : vs) \to vs$	$\text{TAIL}_{\to}$
$\textbf{if}\ true\ \textbf{then}\ e\ \textbf{else}\ e' \to e$	$\text{IF1}_{\to}$
$\textbf{if}\ false\ \textbf{then}\ e\ \textbf{else}\ e' \to e'$	$\text{IF2}_{\to}$
$\mu x.e \to e\,[\mu x.e/x]$	$\text{REC}_{\to}$

The $\text{ADD}_{\to}$ rule states that an addition of two natural numbers can be performed by adding together the two numbers. Since we are counting the number of small-step rules as an approximation of the time usage, this means that performing the actual

addition will be counted as one step, no matter what the values of the numbers are. The NULL1$_{\rightarrow}$ rule states that calling *null* on an empty list reduces to *true*, and correspondingly NULL2$_{\rightarrow}$ states that calling *null* on a list of the form $v : vs$ reduces to *false*. If the condition on the if-statement is *true* then the whole expression reduces to the **then** branch (rule IF1$_{\rightarrow}$), otherwise if it is *false* then expression reduces to the **else** branch (rule IF2$_{\rightarrow}$). The semantics of the recursion operator $\mu x.e$ is to reduce e with all free occurrences of x replaced by $\mu x.e$.

2.10 Interpreter

In order to make traces for evaluating functions less error-prone and time consuming, we implement an interpreter in Haskell. The extended language syntax can be represented as the following Haskell datatype, in which variables are chosen to be integers in order to simplify the process of choosing fresh variable names:

```
data Expr = Var Int | Abs Int Expr | App Expr Expr
          | ConstI Int | ConstB Bool | Nil | Cons Expr Expr
          | Add Expr Expr | Null Expr | Head Expr | Tail Expr
          | If Expr Expr Expr | Rec Int Expr
```

The small-step rules are implemented in the function *beta*, which takes an expression and tries to apply a rule, producing a result of type *Maybe Expr*. If the structure of the expression matches the left-hand side of one of the rules it returns *Just* of the right-hand side, otherwise it returns *Nothing*:

```
beta                                :: Expr → Maybe Expr
beta (App (Abs x e) e')
  | isValue e'                      = Just (subst e e' x)
  | otherwise                       = Nothing
beta (Add (ConstI x) (ConstI y))    = Just (ConstI (x + y))
beta (Null Nil)                     = Just (ConstB True)
beta (Null (Cons x xs))
  | (isValue x) ∧ (isValue xs)      = Just (ConstB False)
```

$$
\begin{aligned}
&\quad \mid otherwise && = Nothing \\
&beta\ (Head\ (Cons\ x\ xs)) \\
&\quad \mid (isValue\ x) \wedge (isValue\ xs) && = Just\ x \\
&\quad \mid otherwise && = Nothing \\
&beta\ (Tail\ (Cons\ x\ xs)) \\
&\quad \mid (isValue\ x) \wedge (isValue\ xs) && = Just\ xs \\
&\quad \mid otherwise && = Nothing \\
&beta\ (If\ (ConstB\ True)\ e\ e') && = Just\ e \\
&beta\ (If\ (ConstB\ False)\ e\ e') && = Just\ e' \\
&beta\ (Rec\ x\ e) && = Just\ (subst\ e\ (Rec\ x\ e)\ x) \\
&beta\ \_ && = Nothing
\end{aligned}
$$

The β-substitution function is implemented as the function *subst*, which has the semantics $subst\ e\ e'\ x = e\,[e'/x]$, and is defined as shown below:

$$
\begin{aligned}
&subst && :: Expr \to Expr \to Int \to Expr \\
&subst\ (Var\ b)\ y\ a \\
&\quad \mid a == b && = y \\
&\quad \mid otherwise && = Var\ b \\
&subst\ (Abs\ b\ x)\ y\ a \\
&\quad \mid a == b && = Abs\ b\ x \\
&\quad \mid \neg\ (elem\ b\ (free\ y)) && = Abs\ b\ (subst\ x\ y\ a) \\
&\quad \mid otherwise && = Abs\ b'\ (subst\ x'\ y\ a) \\
&\qquad\qquad && \textbf{where} \\
&\qquad\qquad && \quad b' = fresh\ (free\ x +\!\!+ free\ y) \\
&\qquad\qquad && \quad x' = subst\ x\ (Var\ b')\ b \\
&subst\ (App\ x1\ x2)\ y\ a && = App\ (subst\ x1\ y\ a)\ (subst\ x2\ y\ a)
\end{aligned}
$$

This definition covers the lambda calculus fragment of the language. There also needs to be cases for each of the other constructs in the language, to define how substitution is propagated through the expression. However, these are straightforward and can be implemented as in the application case above.

The definition of *subst* requires two auxiliary functions, *free* and *fresh*. The func-

tion *free* returns a list of all free variables in an expression, and *fresh*, when applied to a list of variables currently in use, generates a new variable for use:

$$
\begin{aligned}
&\mathit{free} && :: \mathit{Expr} \to [\mathit{Int}] \\
&\mathit{free}\ (\mathit{Var}\ a) && = [a] \\
&\mathit{free}\ (\mathit{Abs}\ a\ x) && = \mathit{remove}\ a\ (\mathit{free}\ x) \\
&\mathit{free}\ (\mathit{App}\ x\ y) && = \mathit{free}\ x +\!\!+ \mathit{free}\ y \\
&\mathit{remove} && :: \mathit{Eq}\ a \Rightarrow a \to [a] \to [a] \\
&\mathit{remove}\ x\ xs && = [y \mid y \leftarrow xs, y \neq x] \\
&\mathit{fresh} && :: [\mathit{Int}] \to \mathit{Int} \\
&\mathit{fresh}\ as && = \mathit{maximum}\ as + 1
\end{aligned}
$$

The definition of *fresh* exploits the representation of variables as integers, by finding the maximum number of the free variables and then adding one to create a new, unused, variable name.

The *step* function takes an expression and performs a single reduction step. If the whole expression matches one of the small-step reduction rules then this is applied, otherwise the *context* function is called to find the appropriate subexpression to reduce:

$$
\begin{aligned}
&\mathit{step} && :: \mathit{Expr} \to \mathit{Expr} \\
&\mathit{step}\ e && = \mathbf{case}\ (\mathit{beta}\ e)\ \mathbf{of} \\
& && \quad (\mathit{Just}\ e') \to e' \\
& && \quad \mathit{Nothing} \to \mathit{context}\ e
\end{aligned}
$$

In turn, the function *context* takes an expression and performs a reduction in the correct context, based on the structure of the expression:

$$
\begin{aligned}
&\mathit{context} && :: \mathit{Expr} \to \mathit{Expr} \\
&\mathit{context}\ (\mathit{App}\ c\ e) && \mid \mathit{isValue}\ c = \mathit{App}\ c\ (\mathit{step}\ e) \\
& && \mid \mathit{otherwise} = \mathit{App}\ (\mathit{step}\ c)\ e \\
&\mathit{context}\ (\mathit{Cons}\ c\ e) && \mid \mathit{isValue}\ c = \mathit{Cons}\ c\ (\mathit{step}\ e) \\
& && \mid \mathit{otherwise} = \mathit{Cons}\ (\mathit{step}\ c)\ e \\
&\mathit{context}\ (\mathit{Add}\ c\ e) && \mid \mathit{isValue}\ c = \mathit{Add}\ c\ (\mathit{step}\ e)
\end{aligned}
$$

$$
\begin{aligned}
&\mid otherwise = Add\ (step\ c)\ e \\
context\ (Null\ c) &= Null\ (step\ c) \\
context\ (Head\ c) &= Head\ (step\ c) \\
context\ (Tail\ c) &= Tail\ (step\ c) \\
context\ (If\ c\ e\ e') &= If\ (step\ c)\ e\ e' \\
context\ \_ &= error\ \texttt{"No context found"}
\end{aligned}
$$

The auxiliary function *isValue* determines if an expression is a value:

$$
\begin{aligned}
isValue &:: Expr \rightarrow Bool \\
isValue\ (Abs\ \_\ \_) &= True \\
isValue\ (ConstI\ \_) &= True \\
isValue\ (ConstB\ \_) &= True \\
isValue\ Nil &= True \\
isValue\ (Cons\ x\ xs) &= isValue\ x \wedge isValue\ xs \\
isValue\ \_ &= False
\end{aligned}
$$

Finally, the function *trace* reduces an expression by repeatedly applying the *step* function until the expression is a value, saving each step of the evaluation to produce a list of expressions, and is defined as follows:

$$
\begin{aligned}
trace &:: Expr \rightarrow [Expr] \\
trace\ e \mid isValue\ e &= [e] \\
\mid otherwise &= e : (trace\ (step\ e))
\end{aligned}
$$

The number of reduction steps is therefore one less than the length of the trace produced, because we wish to count the transitions between expressions. The maximum expression size generated can be calculated by mapping a *size* function over each expression in the trace, and taking the maximum value of the resulting list:

$$
\begin{aligned}
numSteps &:: Expr \rightarrow Int \\
numSteps &= (\lambda x \rightarrow x - 1) \circ length \circ trace \\
maxExpression &:: Expr \rightarrow Int \\
maxExpression &= maximum \circ map\ size \circ trace
\end{aligned}
$$

Calculating the size of an expression, by counting constructors, can be defined over

the structure of the language by simply adding one and calling *size* recursively on any expression arguments in each constructor case:

$$
\begin{aligned}
&\mathit{size} && :: \mathit{Expr} \to \mathit{Int} \\
&\mathit{size}\ (\mathit{Var}\ x) && = 1 \\
&\mathit{size}\ (\mathit{Abs}\ x\ e) && = 1 + \mathit{size}\ e \\
&\mathit{size}\ (\mathit{App}\ f\ e) && = 1 + \mathit{size}\ f + \mathit{size}\ e \\
&\mathit{size}\ (\mathit{ConstI}\ x) && = 1 \\
&\mathit{size}\ (\mathit{ConstB}\ x) && = 1 \\
&\mathit{size}\ \mathit{Nil} && = 1 \\
&\mathit{size}\ (\mathit{Cons}\ x\ xs) && = 1 + \mathit{size}\ x + \mathit{size}\ xs \\
&\mathit{size}\ (\mathit{Add}\ x\ y) && = 1 + \mathit{size}\ x + \mathit{size}\ y \\
&\mathit{size}\ (\mathit{Null}\ xs) && = 1 + \mathit{size}\ xs \\
&\mathit{size}\ (\mathit{Head}\ xs) && = 1 + \mathit{size}\ xs \\
&\mathit{size}\ (\mathit{Tail}\ xs) && = 1 + \mathit{size}\ xs \\
&\mathit{size}\ (\mathit{If}\ b\ e\ e') && = 1 + \mathit{size}\ b + \mathit{size}\ e + \mathit{size}\ e' \\
&\mathit{size}\ (\mathit{Rec}\ x\ e) && = 1 + \mathit{size}\ e
\end{aligned}
$$

2.11 Example

By way of an example, we show how the machinery introduced in this chapter can be used to calculate the time performance of a simple summation function for lists.

Summing a list

The *sum* function can be expressed in the language as follows:

$$\mathit{sum} = \mu\ \mathit{sum}'.\lambda xs.\mathbf{if}\ \mathit{null}\ xs\ \mathbf{then}\ 0\ \mathbf{else}\ \mathit{add}\ (\mathit{head}\ xs)\ (\mathit{sum}'\ (\mathit{tail}\ xs))$$

Translating this definition into a value of type *Expr* and calling the interpreter on the *sum* function applied to the list $[1, 2]$ gives the number of reduction steps as

18, and the maximum expression size as 35. The *sum* function requires considerably more reduction steps than the original function would, because the language doesn't have some of the convenient features of Haskell such as pattern matching. Hence the argument list has to be explicitly tested to check whether it is empty or not, whereas in the original Haskell definition this is done implicitly, and not counted as a step.

Calculating time usage

The time usage for *sum* can be expressed formally, as shown in section 1.4, as a function on the length of the argument list. First of all, we write $\mid e \mid$ for the number of steps required to evaluate e, defined as follows:

$$\mid e \mid = n \quad = \quad \exists v.e \xrightarrow{n} v$$

That is, an expression e can be evaluated in n steps if there exists a value v such that $e \xrightarrow{n} v$. The function *sum xs* has linear complexity if the number of steps can be expressed as a function f of the form $a * (length\ xs) + b$. The values of a and b can be calculated by starting with the following specification:

$$\forall xs.f(length\ xs) = \mid sum\ xs \mid$$

The definition of f can be calculated by first expanding the definition of $\mid e \mid$ and then proceeding by induction on the list xs:

$$\begin{array}{ll} & f\ (length\ xs) = \mid sum\ xs \mid \\ = & \quad \{\ \text{definition of} \mid e \mid\ \} \\ & \exists v.sum\ xs \xrightarrow{f(length\ xs)} v \end{array}$$

Case : $[\,]$

$$\begin{array}{ll} & sum\ [\,] \\ = & \quad \{\ \text{definition of } sum\ \} \end{array}$$

$(\mu\ sum'.\lambda xs.\textbf{if}\ (null\ xs)\ \textbf{then}\ 0\ \textbf{else}\ add\ (head\ xs)\ (sum'\ (tail\ xs)))\ [\,]$

$\rightarrow$ { $\text{REC}_{\rightarrow}$ }

$(\lambda xs.\textbf{if}\ (null\ xs)\ \textbf{then}\ 0\ \textbf{else}\ add\ (head\ xs)\ (sum\ (tail\ xs)))\ [\,]$

$\rightarrow$ { $\beta_{\rightarrow}$ }

$\textbf{if}\ (null\ [\,])\ \textbf{then}\ 0\ \textbf{else}\ add\ (head\ [\,])\ (sum\ (tail\ [\,]))$

$\rightarrow$ { $\text{NULL1}_{\rightarrow}$ }

$\textbf{if}\ true\ \textbf{then}\ 0\ \textbf{else}\ add\ (head\ [\,])\ (sum\ (tail\ [\,]))$

$\rightarrow$ { $\text{IF1}_{\rightarrow}$ }

0

Hence, we have $sum\ [\,] \xrightarrow{4} 0$.

Case : $(x : xs)$

$sum\ (x : xs)$

$=$ { definition of sum }

$(\mu\ sum'.\lambda xs.\textbf{if}\ (null\ xs)\ \textbf{then}\ 0\ \textbf{else}\ add\ (head\ xs)\ (sum'\ (tail\ xs)))\ (x : xs)$

$\rightarrow$ { $\text{REC}_{\rightarrow}$ }

$(\lambda xs.\textbf{if}\ (null\ xs)\ \textbf{then}\ 0\ \textbf{else}\ add\ (head\ xs)\ (sum\ (tail\ xs)))\ (x : xs)$

$\rightarrow$ { $\beta_{\rightarrow}$ }

$\textbf{if}\ (null\ (x : xs))\ \textbf{then}\ 0\ \textbf{else}\ add\ (head\ (x : xs))\ (sum\ (tail\ (x : xs)))$

$\rightarrow$ { $\text{NULL2}_{\rightarrow}$ }

$\textbf{if}\ false\ \textbf{then}\ 0\ \textbf{else}\ add\ (head\ (x : xs))\ (sum\ (tail\ (x : xs)))$

$\rightarrow$ { $\text{IF2}_{\rightarrow}$ }

$add\ (head\ (x : xs))\ (sum\ (tail\ (x : xs)))$

$\rightarrow$ { $\text{HEAD}_{\rightarrow}$ }

$add\ x\ (sum\ (tail\ (x : xs)))$

$\rightarrow$ { $\text{TAIL}_{\rightarrow}$ }

$add\ x\ (sum\ xs)$

$\xrightarrow{f(length\ xs)}$ { inductive assumption, $add\ v\ C$ context }

$add\ x\ m$

$\rightarrow$ { $\text{ADD}_{\rightarrow}$, $x, m \in \mathbb{N}$ }

$$x + m$$

Hence, we have $\exists m.sum\ (x : xs) \overset{7+f(length\ xs)}{\rightarrow} x + m$. These two results can be used to calculate the definition of the function f, as follows:

$$
\begin{array}{ll}
 & sum\ [\,] \overset{4}{\rightarrow} 0 \quad \text{and} \\
 & \exists m.sum\ (x : xs) \overset{7+f(length\ xs)}{\rightarrow} x + m \\
= & \quad \{\ \text{definition of } e \overset{n}{\rightarrow} e'\ \} \\
 & f\ (length\ [\,]) = 4 \\
 & f\ (length\ (x : xs)) = 7 + f\ (length\ xs) \\
= & \quad \{\ \text{definition of } length\ \} \\
 & f\ 0 = 4 \\
 & f\ (1 + length\ xs) = 7 + f\ (length\ xs) \\
= & \quad \{\ \text{generalising } length\ xs \text{ to } n\ \} \\
 & f\ 0 = 4 \\
 & f\ (n + 1) = 7 + f\ (n) \\
= & \quad \{\ \text{induction over } n\ \} \\
 & f\ n = 4 + 7\ n
\end{array}
$$

In conclusion, we have shown that the evaluating the function *sum* applied to a list argument *xs* requires the following number of steps:

$$|\ sum\ xs\ | = 4 + 7 * (length\ xs)$$

This shows that the *sum* function is still linear, because only the constants have changed from the original *sum* function.

Big-step rules

The big-step rules presented earlier for the lambda calculus can be extended to incorporate the new language structures as follows:

$$\frac{}{v \Downarrow^0 v}\ \text{VAL}_{\Downarrow} \qquad \frac{f \Downarrow^a \lambda x.e' \qquad e \Downarrow^b v' \qquad e'\,[v'\,/\,x] \Downarrow^c v}{f\ e \Downarrow^{a+b+c+1} v}\ \text{APP}_{\Downarrow}$$

$$\frac{e[\mu\ x.e/x] \Downarrow^a v}{\mu\ x.e \Downarrow^{a+1} v}\ \text{REC}_{\Downarrow} \qquad \frac{x \Downarrow^a n \qquad y \Downarrow^b m}{add\ x\ y \Downarrow^{a+b+1} (n+m)}\ \text{ADD}_{\Downarrow}$$

$$\frac{xs \Downarrow^a [\,]}{null\ xs \Downarrow^{a+1} true}\ \text{NULL-1}_{\Downarrow} \qquad \frac{xs \Downarrow^a v : vs}{null\ xs \Downarrow^{a+1} false}\ \text{NULL-2}_{\Downarrow}$$

$$\frac{e \Downarrow^a (v : vs)}{head\ e \Downarrow^{a+1} v}\ \text{HEAD}_{\Downarrow} \qquad \frac{e \Downarrow^a (v : vs)}{tail\ e \Downarrow^{a+1} vs}\ \text{TAIL}_{\Downarrow}$$

$$\frac{p \Downarrow^a true \qquad e \Downarrow^b v}{\textbf{if}\ p\ \textbf{then}\ e\ \textbf{else}\ e' \Downarrow^{a+b+1} v}\ \text{IF-1}_{\Downarrow} \qquad \frac{p \Downarrow^a false \qquad e' \Downarrow^b v}{\textbf{if}\ p\ \textbf{then}\ e\ \textbf{else}\ e' \Downarrow^{a+b+1} v}\ \text{IF-2}_{\Downarrow}$$

$$\frac{x \Downarrow^a v \qquad xs \Downarrow^b vs}{x : xs \Downarrow^{a+b} v : vs}\ \text{CONS}_{\Downarrow}$$

Unlike the other rules, the $\text{CONS}_{\Downarrow}$ rule do not add to the step count because it just shows how an expression is to be evaluated, without performing any computation.

Calculating time usage

The time usage for the *sum* function can also be calculated using these rules, by first redefining $\mid e \mid$ to be the number of steps required in the big step semantics:

$$\mid e \mid = n \quad \Leftrightarrow \quad \exists v.e \Downarrow^n v$$

The time requirements for *sum* can then be calculated, as previously, by expanding the definition of $\mid e \mid$ and then proceeding by induction over the list xs:

$$\begin{array}{ll} & f\ (length\ xs) = \mid sum\ xs \mid \\ = & \quad \{\ \text{definition of} \mid e \mid\ \} \\ & \exists v.sum\ xs \Downarrow^{f(length\ xs)} v \end{array}$$

Case : $[\,]$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{true \Downarrow^0 true}\,\text{VAL}_{\Downarrow}}{null\ [\,] \Downarrow^1 true}\,\text{NULL-1}_{\Downarrow} \qquad \dfrac{}{0 \Downarrow^0 0}\,\text{VAL}_{\Downarrow}}{\textbf{if}\ (null\ [\,])\ \textbf{then}\ 0\ \textbf{else}\ add\ (head\ xs)\ (sum\ (tail\ [\,])) \Downarrow^2 0}\,\text{IF-1}_{\Downarrow}}{(\lambda xs.\textbf{if}\ (null\ xs)\ \textbf{then}\ 0\ \textbf{else}\ add\ (head\ xs)\ (sum\ (tail\ xs)))\ [\,] \Downarrow^3 0}\,\text{APP}_{\Downarrow}}{sum\ [\,] \Downarrow^4 0}\,\text{REC}_{\Downarrow}$$

Case : $(v : vs)$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\lambda xs.e' \Downarrow^0 \lambda xs.e'}\,\text{VAL}_{\Downarrow}}{sum \Downarrow^1 \lambda xs.e'}\,\text{REC}_{\Downarrow} \qquad \dfrac{\dfrac{}{v : vs \Downarrow^0 v : vs}\,\text{VAL}_{\Downarrow}}{null\ (v : vs) \Downarrow^1 false}\,\text{NULL-2}_{\Downarrow} \qquad \dfrac{(1)}{\Downarrow^{n+3} v + m}\,\text{ADD}_{\Downarrow}}{e'\ [v : vs\ /\ xs] \Downarrow^{n+5} v + m}\,\text{IF-1}_{\Downarrow}}{\textbf{where}\ e' = \textbf{if}\ (null\ xs)\ \textbf{then}\ 0\ \textbf{else}\ add\ (head\ xs)\ (sum\ (tail\ xs))}\,\text{APP}_{\Downarrow}}{sum\ (x : xs) \Downarrow^{n+7} v + m}$$

The subtree (1) expands to the following:

$$\dfrac{\textsc{head}_{\Downarrow}\ \dfrac{\textsc{val}_{\Downarrow}\ \dfrac{}{v : vs \Downarrow^{0} v : vs}}{head\ (v : vs) \Downarrow^{1} v} \qquad \dfrac{\textsc{val}_{\Downarrow}\ \dfrac{}{v : vs \Downarrow^{0} v : vs}}{tail\ (v : vs) \Downarrow^{1} vs}\ \textsc{tail}_{\Downarrow}\ \dfrac{}{sum\ vs \Downarrow^{n} m}\ \textsc{Hyp}}{sum\ (tail\ (v : vs)) \Downarrow^{n+1} m}\ \textsc{app}_{\Downarrow}$$

The result of the derivations are that:

$$sum\ [\,] \Downarrow^{4} 0$$
$$sum\ (v : vs) \Downarrow^{7+f(length\ vs)} x + m$$

As expected, the resulting time usage is the same as for the small-step semantics.

2.12 Summary

This chapter has approached the problem of space and time analysis of functional programs in a conventional way, by defining an operational semantics that counts β-reductions and primitive function applications. Two different operational semantics were presented, showing how the view of a reduction step can be abstracted. These two semantics were shown to be equivalent and time requirements were calculated for the simple example of *sum* using both semantics.

The work in this chapter was carried out as a preliminary study, and was used to identify an area that further research could contribute to. Previous work has concentrated on producing bounds from generally recursive programs by first converting them to closed-forms, as in [6, 18]. Producing these closed-form solutions can be simplified by expressing the examples in a more structured way, and introducing recursion operators more specific to the data structures in the language. As an example, if we are interested in studying the space and time behaviour of functions over lists then we could define the *fold* operator over lists and express the examples using this operator, instead of μ. This approach makes it easier to express space and time requirements as a function over the size of the input, because the resulting functions are already

in a more structured form, and also facilitates the use of program transformation techniques such as fusion [40].

The other area identified was that of the accuracy of the steps that were being counted. As already mentioned, measuring β-reductions is an approximation, because substituting into an expression may take arbitrarily long. The remaining chapters addresses this concern, by proposing instead to count transitions in an abstract machine. As will be seen, this method also provides a structured way to measure the space usage, by measuring the data structures in the machine.

Chapter 3

Abstract machines

In the previous chapter a small-step semantics was used to estimate the time requirements for evaluating expressions. The reasoning for this was that at the level of the small-step semantics each evaluation step was visible, as opposed to a big step semantics where evaluation appears to occur in one step. Counting these small steps was used to measure the time usage of evaluation. However, there was no justification that the steps were in any way related to how the program would be executed on an actual machine. Moreover, the semantics presented did not show the space usage of evaluation, because the small-step rules did not expose any of the underlying data structures required, for example in dealing with function calls.

One approach to addressing these issues would be to consider the behaviour of a real machine, such as the *Spineless Tagless G-machine* [41] that is used in the Glasgow Haskell Compiler. However, for the purposes of this thesis we take a different approach. First of all, we abstract from the details of a low-level machine implementation, in order that the information we derive is of a higher-level and is not tied to a particular machine. Secondly, this would require taking account of the many optimisations that are applied in a real compiler such as GHC, which is an interesting and important topic, but would be a subject for a thesis in its own right. Finally, these optimisations would also make it difficult to reflect time and space information back to a higher-level, which is one of our principle aims. Our thesis is that the notion of an abstract machine that corresponds to an interpreter, rather than a compiler,

provides an appropriate compromise between abstraction and accuracy.

3.1 Introduction

An abstract machine is a model of computation for executing programs in a given language, and consists of a set of states S with a transition relation $\rightarrow \subseteq S \times S$. These transitions are defined using rewrite rules that make explicit the step-by-step process by which evaluation is performed [42].

To introduce this concept we will first look at a simple example, that is inspired by Hutton and Wright [42]. Consider a simple language in which programs consist of a sequence of operations, which either involve pushing an integer onto a stack, or adding the top two integers on the stack, where the stack is a list of integers. This language can be represented in Haskell by the following datatypes:

$$\begin{aligned}&\textbf{type } \mathit{Prog} &&= [\mathit{Op}]\\ &\textbf{data } \mathit{Op} &&= \mathit{PUSH}\ \mathit{Int} \mid \mathit{ADD}\\ &\textbf{type } \mathit{Stack} &&= [\mathit{Int}]\end{aligned}$$

An abstract machine to execute this language can be defined using the following two rewrite rules, where a state in the machine is a pair of type $(\mathit{Prog}, \mathit{Stack})$:

$$\begin{aligned}(\mathit{PUSH}\ n : \mathit{ops}, s) &\rightarrow (\mathit{ops}, n : s)\\ (\mathit{ADD} : \mathit{ops}, n : m : s) &\rightarrow (\mathit{ops}, n + m : s)\end{aligned}$$

The first rule states that pushing an integer puts that integer on to the top of the stack, and the second rule states that performing an add operation first removes the top two integers from the stack, adds them together, and finally puts this new value back on top of the stack.

The machine is run by first loading it with the program to be evaluated, and an empty stack. The rewrite rules are then repeatedly applied until no longer possible. Finally the machine unloaded, in this case by returning the value on the top of the stack. The abstract machine may be expressed as a function in Haskell as follows:

$$
\begin{aligned}
&\mathit{run} && :: \mathit{Prog} \to \mathit{Int} \\
&\mathit{run} && = \mathit{unload} \circ \mathit{exec} \circ \mathit{load} \\
&&& \quad \textbf{where} \\
&&& \quad\quad \mathit{load}\ p && = (p, [\,]) \\
&&& \quad\quad \mathit{unload}\ (p, s) && = \mathit{head}\ s \\
&\mathit{exec} && :: (\mathit{Prog}, \mathit{Stack}) \to (\mathit{Prog}, \mathit{Stack}) \\
&\mathit{exec}\ (\mathit{PUSH}\ n : \mathit{ops}, s) && = \mathit{exec}\ (\mathit{ops}, n : s) \\
&\mathit{exec}\ (\mathit{ADD} : \mathit{ops}, n : m : s) && = \mathit{exec}\ (\mathit{ops}, n + m : s) \\
&\mathit{exec}\ (p, s) && = (p, s)
\end{aligned}
$$

Note that the implementation of an abstract machine in a functional language manifests itself as a *first-order* and *tail-recursive* function — first-order because it passes data structures around as arguments, rather than passing functions, and tail-recursive because each right-hand-side of a function definition consists of a direct function call. This latter term is explained in more detail below.

Tail recursion

Tail-recursion is a property of how the body of a function is structured. As an example, consider the following definition for the factorial function:

$$
\begin{aligned}
&\mathit{fact} && :: \mathit{Int} \to \mathit{Int} \\
&\mathit{fact}\ 0 && = 1 \\
&\mathit{fact}\ n && = n * \mathit{fact}\ (n - 1)
\end{aligned}
$$

This definition is not tail-recursive because in the n case for *fact*, after the recursive call to calculate $\mathit{fact}\ (n - 1)$ returns, there is still the multiplication with n to be performed. In contrast, the following alternative definition uses an accumulator:

$$
\begin{aligned}
&\mathit{fact} = \mathit{factAcc}\ 1 \\
&\quad \textbf{where} \\
&\quad\quad \mathit{factAcc}\ a\ 0 = a \\
&\quad\quad \mathit{factAcc}\ a\ n = \mathit{factAcc}\ (n * a)\ (n - 1)
\end{aligned}
$$

This definition is now tail-recursive because in the n case for *factAcc*, after the recursive call to *factAcc*, the value it computed is immediately returned.

Tail recursion has two properties that are relevant to studying time and space behaviour. The first is that it is simpler to implement than other forms of recursion, because there is no need to allocate storage on the stack at run-time for nested function calls. This means that there is no hidden overhead in executing a tail-recursive program that cannot be seen at the program level.

The second property is that tail-recursion corresponds to iteration in imperative languages, and this would allow the use of conventional imperative techniques for measuring space and time properties. For example, the accumulator version of factorial above is equivalent to the following imperative version:

```
int fact (int n) {
  int a = 1;              (1)
  while (n > 0) {         (n)
    a = n*a;              (4n)
    n = n-1;              (3n)
  }
  return a;               (1)
}
```

As in section 1.2, by considering the number of times the loop is executed and summing the cost of each line of the program (shown in parentheses above), we can observe that the time requirements to compute the result of *fact* n are $8n + 2$.

Comparison to small-step semantics

The transitions of an abstract machine may appear to be similar to the small-step semantics presented earlier, in that they both share the same syntax, $\rightarrow$, but there is an important distinction. The rules in the small-step semantics may have preconditions, which need to be satisfied in order for the rule to be applied, and in order

to do this an implicit execution stack is required. For example, when evaluating an expression of the form $f\ e$, we need to apply the application rule:

$$\frac{f \to f'}{f\ e \to f'\ e}$$

However, to do this we first of all need to find an f' such that $f \to f'$. If f is itself an application, then this will continue, generating an implicit stack of computations until a rule is applied that has no premise, and then the stack can be unwound.

In contrast, the transitions of an abstract machine are simple rewrite rules, and therefore choosing which transition to apply next is based solely on the current state, i.e. the values of its data structures at that point. In this way, abstract machines provide a more realistic model of evaluation than small-step semantics, because all the additional data structures required for evaluation are made explicit. For this reason, we choose to use abstract machines as a more appropriate level of abstraction for measuring space and time information.

Virtual machine

A related concept to the notion of an abstract machine is that of a virtual machine. While some authors use these terms interchangeably, for our purposes it is important to clarify the distinction in their meaning. In particular, an abstract machine operates directly on the expressions in the original language, whereas a virtual machine operates on a compiled version of such expressions, and thereby requires a separate instruction set [43]. For example, suppose that we define a simple language of arithmetic expression trees, with integers at the leaves and addition at the nodes:

$$\textbf{data}\ \mathit{Expr} = \mathit{Val}\ \mathit{Int} \mid \mathit{Add}\ \mathit{Expr}\ \mathit{Expr}$$

Expressions in this language may be compiled into a list of operations as presented earlier by simply performing a post-order flattening of the tree:

$$\mathit{compile} :: \mathit{Expr} \to \mathit{Prog}$$

$$\begin{aligned} \mathit{compile}\ (\mathit{Val}\ n) &= [\mathit{PUSH}\ n] \\ \mathit{compile}\ (\mathit{Add}\ x\ y) &= \mathit{compile}\ x +\!\!+ \mathit{compile}\ y +\!\!+ [\mathit{ADD}] \end{aligned}$$

The abstract machine presented in section 3.1 is now a virtual machine for the language *Expr*, showing that whether a machine is considered to be abstract or virtual depends on the context.

3.2 Deriving machines

Traditionally, abstract machines have been designed independently, and then proved equivalent to a high-level semantics. However, this has meant that the intuition behind the workings of the machine is often not clear, as we now show.

The SECD machine

Landin's SECD machine [44] was the first abstract machine for executing the lambda calculus under call-by-value evaluation. The SECD machine may also be described as a virtual machine, using the lambda calculus as an intermediate language. The name SECD stands for the four data structures required by the machine: each state is a dump, which is either empty or consists of a four-tuple (S, E, C, D), where S is a stack, E an environment, C the control (or code) and D is a further dump. The stack is a sequence of closures, where a closure is an expression paired with an environment. The environment consists of a set of (variable, closure) pairs and the control is a stack of lambda expressions and a special token, *APP*. The transition relation is defined between dumps, according to the following rules:

$$\begin{aligned} (cl : S, E, \epsilon, (S', E', C', D')) &\rightarrow (cl : S', E', C', D') \\ (S, E, x : C, D) &\rightarrow (\mathit{lookup}\ x\ E : S, E, C, D) \\ (S, E, (\lambda x.e) : C, D) &\rightarrow ((\lambda x.e, E) : S, E, C, D) \\ ((\lambda x.e, E') : e' : S, E, \mathit{APP} : C, D) &\rightarrow (\epsilon, E'[x \mapsto e'], e : \epsilon, (S, E, C, D)) \\ (S, E, fe : C, D) &\rightarrow (S, E, e : f : \mathit{APP} : C, D) \end{aligned}$$

The above rules illustrate that abstract machines are not always as straightforward as the example presented in section 3.1. In particular, it is difficult to observe from the transition rules above which evaluation strategy is being used, and what the justification is for the data structures and transitions. Indeed, although the SECD machine was originally presented in 1964, it is only recently that the rationale for its design has been articulated in a formal manner, by Danvy [45].

Deriving machines

Danvy *et al* have also shown how to approach the problem of designing abstract machines from the opposite angle, by deriving machines from evaluators [43, 46]. This is a systematic process, achieved by performing successive program transformations on the evaluator. A particular advantage of this approach is that because the abstract machine has been derived directly from the evaluator, there is now no need to prove that the machine correctly implements the evaluator.

The next section will show in detail how an abstract machine can be *calculated* from an evaluator. The calculation requires in particular two program transformation techniques, transformation in to continuation-passing style, and defunctionalization, and we begin by reviewing these two techniques in turn.

Continuation-passing style

A function that is not tail-recursive can be made tail-recursive by converting it into *continuation-passing style* [47]. There are two contrasting views of what a continuation is [48]: either it represents an evaluation context, or the rest of a computation. In this case we take the second view, in which a continuation is a function that is passed an intermediate value and uses this value to compute the final result.

If a function f has type $a \to b$, then the continuation required will be of type $b \to b$. The function f can be transformed into continuation-passing style by defining a new function f' that takes the continuation as an additional argument, and therefore has type $a \to (b \to b) \to b$. The desired behaviour of f' can be characterised by the

specification $f'\ a\ c = c\ (f\ a)$, which states that the new function f' behaves in the same way as the continuation applied to the function f.

As an example, we revisit the factorial function:

$$\begin{aligned} &fact &&:: Int \to Int \\ &fact\ 0 &&= 1 \\ &fact\ n &&= n * fact\ (n-1) \end{aligned}$$

As discussed earlier, this definition is not tail-recursive because the multiplication in the recursive call can only be performed after the call to $fact\ (n-1)$ has returned. The factorial function can be converted into continuation-passing style by defining a new function $fact'$ that takes a continuation, which can be calculated from the specification $fact'\ n\ c = c\ (fact\ n)$, by induction on the natural number n:

Case : 0

$$\begin{aligned} & fact'\ 0\ c \\ = & \quad \{\ \text{specification}\ \} \\ & c\ (fact\ 0) \\ = & \quad \{\ \text{definition of } fact\ \} \\ & c\ 1 \end{aligned}$$

Case : n

$$\begin{aligned} & fact'\ n\ c \\ = & \quad \{\ \text{specification}\ \} \\ & c\ (fact\ n) \\ = & \quad \{\ \text{definition of } fact\ \} \\ & c\ (n * fact\ (n-1)) \\ = & \quad \{\ \text{reverse beta-reduction}\ \} \\ & (\lambda x \to c\ (n * x))\ (fact\ (n-1)) \\ = & \quad \{\ \text{inductive assumption}\ \} \\ & fact'\ (n-1)\ (\lambda x \to c\ (n * x)) \end{aligned}$$

In conclusion, we have calculated the following definition for $fact'$, from which the original factorial function $fact$ can be recovered by supplying the identity function for the initial continuation:

$$
\begin{aligned}
&fact' && :: Int \to (Int \to Int) \to Int \\
&fact'\ 0\ c && = c\ 1 \\
&fact'\ n\ c && = fact'\ (n-1)\ (\lambda x \to c\ (n * x)) \\
&fact\ n && = fact'\ n\ id
\end{aligned}
$$

The resulting definition of factorial is now tail-recursive, in that the body of the n case of $fact'$ consists of a direct recursive call.

Defunctionalization

The other program transformation we require is *defunctionalization* [47], which is a technique for eliminating the use of functions as arguments in programs. This technique is based upon the observation we don't usually need the entire function-space of possible argument functions, because only a few forms of such functions are actually used in practice. Hence, we can represent the functions that we actually need using a datatype, rather than using the actual functions themselves. In this section, we briefly review defunctionalization, using the example of a hash table.

A hash table is an abstract datatype that associates keys to objects and has two functions, *get* and *put*, which respectively retrieve and add an object to the table. A functional programmer implementing this data structure might take a higher-order approach, using a hash function that maps keys of type k to objects of type a:

$$\textbf{type}\ Hash\ k\ a = k \to a$$

An initial hash function can be defined that maps all keys to undefined:

$$
\begin{aligned}
&hash :: Eq\ k \Rightarrow Hash\ k\ a \\
&hash = \lambda k \to \bot
\end{aligned}
$$

The *get* function, which given a key returns the object that it maps to, can be defined by simply applying the hash function to the key:

$$
\begin{aligned}
&\mathit{get} && :: \mathit{Eq}\ k \Rightarrow \mathit{Hash}\ k\ a \to k \to a \\
&\mathit{get}\ h\ k && = h\ k
\end{aligned}
$$

A new object can be put in the table under a key by creating a new association between the key and object. This is achieved by creating a new hash function, which takes a key and if it matches the new association key returns the object, and otherwise applies the old hash function to the key. This can be defined as follows:

$$
\begin{aligned}
&\mathit{put} && :: \mathit{Eq}\ k \Rightarrow \mathit{Hash}\ k\ a \to k \to a \to \mathit{Hash}\ k\ a \\
&\mathit{put}\ h\ k\ x && = \lambda k' \to \textbf{if}\ k == k'\ \textbf{then}\ x\ \textbf{else}\ h\ k'
\end{aligned}
$$

These definitions are natural in a functional language, but what isn't obvious is the space and time behaviour of the *put* and *get* functions with this implementation. The space requirements for functions is more apparent when they only involve first-order data structures. A function may be made first-order by defunctionalizing it.

At present, the hash table is a function from keys to objects, but the whole function space is not required, because the hash functions are only created in two different ways: one to make the initial hash table, and another to extend the table when a new object is put into it. The first step in defunctionalization is to locate all places where functions are constructed, and define a new datatype that models these forms of functions. Each constructor of the new data structure Hash' takes as arguments any free variables required:

$$\textbf{data}\ \mathit{Hash}'\ k\ a = \mathit{H1} \mid \mathit{H2}\ k\ a\ (\mathit{Hash}'\ k\ a)$$

In the Hash' datatype above, the constructor *H1* represents the initial hash table ($\lambda k \to \bot$). No arguments are required in this case since there are no free variables in this function. A constructor *H2* is also required for the function $\lambda k' \to \textbf{if}\ k == k'\ \textbf{then}\ x\ \textbf{else}\ h\ k'$ created in *put*. This function has three free variables, a key k, object x and another hash table h, which are taken as arguments to *H2*. The meaning of values of type $\mathit{Hash}'\ k\ a$ is as functions of type $\mathit{Hash}\ k\ a$, and we define a function *apply* that recovers the functionality of each constructor accordingly:

$$
\begin{aligned}
&\mathit{apply} && :: \mathit{Eq}\ k \Rightarrow \mathit{Hash}'\ k\ a \to \mathit{Hash}\ k\ a \\
&\mathit{apply}\ \mathit{H1} && = \lambda k \to \bot
\end{aligned}
$$

$$apply\ (H2\ k\ x\ h') = \lambda k' \rightarrow \textbf{if}\ k == k'\ \textbf{then}\ x\ \textbf{else}\ apply\ h'\ k'$$

We can construct new versions of the functions *hash*, *put* and *get* that work on the new data structure h' by forming specifications that substitute $apply\ h'$ for h. In this case we form the specifications $apply\ hash' = hash$, $get'\ h'\ k = get\ (apply\ h')\ k$ and $put'\ h'\ k\ x = put\ (apply\ h')\ k\ x$, and the result is the following functions:

$$\begin{aligned}
hash' &:: Eq\ k \Rightarrow Hash'\ k\ a \\
hash' &= H1 \\
get' &:: Eq\ k \Rightarrow Hash'\ k\ a \rightarrow k \rightarrow a \\
get'\ h'\ k &= apply\ h'\ k \\
put' &:: Eq\ k \Rightarrow Hash'\ k\ a \rightarrow k \rightarrow a \rightarrow Hash'\ k\ a \\
put'\ h'\ k\ x &= H2\ k\ x\ h'
\end{aligned}$$

Modifying the original functions in this way has made it easier to see their space and time usage. In terms of the space requirements, we can now see that the hash table is essentially just a list of key-object pairs, because the $Hash'\ k\ a$ data structure is equivalent to the list structure $[(k, a)]$. We can also see that there is a certain amount of space-inefficiency, in that when a key is re-associated to a new object, the original mapping is not forgotten but remains in the list and becomes unreachable. Therefore the space required to store the table is proportional to the number of *put* commands there have been, and not just the number of unique keys in the table.

Defunctionalizing has also made the time requirements of the hash table functions more apparent. Inserting an object into the table is a constant time operation, because the key-object pair is just prepended to the list of entries. In the original definition of *put*, the hash function was simply applied to the key to return the associated object. In the defunctionalized version, however, we can see that this application actually requires a list to be searched, so this function will have worst-case time requirements proportional to the length of the list.

3.3 Simple language

Now the concepts of continuation-passing style and defunctionalization have been introduced, we will show how these concepts can be applied to an evaluator in order to transform it into an abstract machine. To start with we will consider the simple language of arithmetic expressions presented earlier, consisting of integers and addition, for which the notion of a value is simply an integer:

$$\textbf{data}\ \mathit{Expr} = \mathit{Val}\ \mathit{Int} \mid \mathit{Add}\ \mathit{Expr}\ \mathit{Expr}$$
$$\textbf{type}\ \mathit{Value} = \mathit{Int}$$

Although this language is not powerful enough to be used to analyse the time requirements of any interesting computations, it will be sufficient to show the derivation process without over-complication. An extended language will be presented in a later section, in order to study the space and time behaviour of some example functions.

Evaluator

The initial evaluator takes an expression and produces its value:

$$\begin{aligned} &\mathit{eval} && :: \mathit{Expr} \to \mathit{Value} \\ &\mathit{eval}\ (\mathit{Val}\ v) && = v \\ &\mathit{eval}\ (\mathit{Add}\ x\ y) && = \mathit{eval}\ x + \mathit{eval}\ y \end{aligned}$$

That is, evaluating an integer value returns that integer, and evaluating an addition evaluates both sides of the addition to an integer and adds them together. The order of evaluation of the arguments to the addition is not specified at this level, but will be determined by the semantics of the underlying language.

Tail-recursive evaluator

Our aim now is to transform the evaluator into an abstract machine. More precisely, we seek to construct a first-order, tail-recursive function in Haskell. The evaluator is already first order, because it does not use functions as arguments or results, but it

is not tail-recursive, because the recursive calls to *eval* are nested within the call to the addition operator. As explained earlier, a function can be made tail-recursive by converting it into continuation-passing style. In this case, the continuation will take an argument of type *Value* and also return a *Value*:

$$\textbf{type } \mathit{Con} = \mathit{Value} \rightarrow \mathit{Value}$$

The new tail-recursive evaluator can be calculated from the original definition for this function starting with the specification:

$$\mathit{evalTail}\ e\ c = c\ (\mathit{eval}\ e)$$

That is, the new evaluator has the same behaviour as simply applying the continuation to the result of the original evaluator. The definition of this function can be calculated by performing induction on the structure of the expression e.

Case : $e = \mathit{Val}\ v$

$$\begin{array}{ll} & \mathit{evalTail}\ (\mathit{Val}\ v)\ c \\ = & \quad \{\ \text{specification}\ \} \\ & c\ (\mathit{eval}\ v) \\ = & \quad \{\ \text{definition of } \mathit{eval}\ \} \\ & c\ v \end{array}$$

Case : $e = \mathit{Add}\ x\ y$

$$\begin{array}{ll} & \mathit{evalTail}\ (\mathit{Add}\ x\ y)\ c \\ = & \quad \{\ \text{specification}\ \} \\ & c\ (\mathit{eval}\ (\mathit{Add}\ x\ y)) \\ = & \quad \{\ \text{definition of } \mathit{eval}\ \} \\ & c\ (\mathit{eval}\ x + \mathit{eval}\ y) \\ = & \quad \{\ \text{reverse } \beta\text{-reduction, abstract over } \mathit{eval}\ x\ \} \\ & (\lambda m \rightarrow c\ (m + \mathit{eval}\ y))\ (\mathit{eval}\ x) \\ = & \quad \{\ \text{inductive assumption for } x\ \} \end{array}$$

$$
\begin{aligned}
& \mathit{evalTail}\ x\ (\lambda m \rightarrow c\ (m + \mathit{eval}\ y)) \\
= & \quad \{\ \text{reverse } \beta\text{-reduction, abstract over } \mathit{eval}\ y\ \} \\
& \mathit{evalTail}\ x\ (\lambda m \rightarrow (\lambda n \rightarrow c\ (m + n))\ (\mathit{eval}\ y)) \\
= & \quad \{\ \text{inductive assumption for } y\ \} \\
& \mathit{evalTail}\ x\ (\lambda m \rightarrow \mathit{evalTail}\ y\ (\lambda n \rightarrow c\ (m + n)))
\end{aligned}
$$

In conclusion, we have calculated the following recursive definition:

$$
\begin{aligned}
& \mathit{evalTail} && :: \mathit{Expr} \rightarrow \mathit{Con} \rightarrow \mathit{Value} \\
& \mathit{evalTail}\ (\mathit{Val}\ v) && c = c\ v \\
& \mathit{evalTail}\ (\mathit{Add}\ x\ y)\ && c = \mathit{evalTail}\ x\ (\lambda m \rightarrow \\
& && \quad \mathit{evalTail}\ y\ (\lambda n \rightarrow \\
& && \quad c\ (m + n)))
\end{aligned}
$$

That is, in the case when the expression is an integer the continuation is simply applied to the integer value. In the addition case, the first argument to the addition is evaluated first, with the result being passed in to a continuation. The second expression argument is then evaluated inside the continuation, with its result being passed in to an inner continuation. Both integer results are added together in the body of this function and the original continuation is applied to the result.

The evaluator is now tail recursive, in that the right-hand side is a direct recursive call. Moreover, in making the evaluator tail-recursive we have introduced an explicit evaluation order: the evaluation of addition now takes place in left-to-right order. Finally, the semantics of the original evaluation function can be recovered by substituting in the identity function for the continuation:

$$\mathit{eval}\ e = \mathit{evalTail}\ e\ \mathit{id}$$

Abstract machine

The function *evalTail* is now tail-recursive, but in achieving this the function has become higher-order, because now we are passing around continuations. The next step is to make the evaluator first-order, which can be achieved by defunctionalization.

At present, the continuations are functions of the type $Value \rightarrow Value$, but the entire function space is not required. In particular, the continuation functions are only created in three specific ways. As in the earlier example, defunctionalization is performed by looking at all places where functions are constructed and replacing them with a new data structure that takes as arguments any free variables required. In this case, the data structure required is as follows:

$$\textbf{data}\ Cont = Top \mid AddL\ Cont\ Expr \mid AddR\ Value\ Cont$$

These constructors — *Top*, *AddL* and *AddR* — represent the initial continuation *id*, and the continuations $(\lambda m \rightarrow evalTail\ y\ (...))$ and $(\lambda n \rightarrow c\ (m + n))$ respectively. The reason for the particular choice of constructor names is that the data structure is the structure of evaluation contexts for the language [48].

It is straightforward to define a function *apply* that converts our representation of a continuation into a real continuation:

$$\begin{aligned}
&apply && :: Cont \rightarrow Con \\
&apply\ Top && = \lambda v \rightarrow v \\
&apply\ (AddR\ m\ c) && = \lambda n \rightarrow apply\ c\ (m + n) \\
&apply\ (AddL\ c\ y) && = \lambda m \rightarrow evalTail\ y\ (apply\ (AddL\ c\ m))
\end{aligned}$$

We now seek to construct a new evaluator, *evalMach*, that behaves in the same way as *evalTail*, except that it uses representations of continuations, rather than real continuations; that is, we require $evalMach\ e\ c = evalTail\ e\ (apply\ c)$. From this specification, the definition of *evalMach* can be calculated by induction on e:

Case : $e = Val\ v$

$$\begin{aligned}
&evalMach\ (Val\ v)\ c \\
=&\quad \{\ \text{specification}\ \} \\
&evalTail\ (Val\ v)\ (apply\ c) \\
=&\quad \{\ \text{definition of } evalTail\ \} \\
&apply\ c\ v
\end{aligned}$$

Case : $e = Add\ x\ y$

$$
\begin{aligned}
& evalMach\ (Add\ x\ y)\ c \\
= & \quad \{\ \text{specification}\ \} \\
& evalTail\ (Add\ x\ y)\ (apply\ c) \\
= & \quad \{\ \text{definition of } evalTail\ \} \\
& evalTail\ x\ (\lambda m \rightarrow evalTail\ y\ (\lambda n \rightarrow apply\ c\ (m + n))) \\
= & \quad \{\ \text{definition of } apply\ \} \\
& evalTail\ x\ (\lambda m \rightarrow evalTail\ y\ (apply\ (AddR\ m\ c))) \\
= & \quad \{\ \text{definition of } apply\ \} \\
& evalTail\ x\ (apply\ (AddL\ c\ y)) \\
= & \quad \{\ \text{inductive assumption, for } x\ \} \\
& evalMach\ x\ (AddL\ c\ y)
\end{aligned}
$$

In conclusion, we have now calculated the following recursive function:

$$
\begin{aligned}
& evalMach && :: Expr \rightarrow Cont \rightarrow Value \\
& evalMach\ (Val\ v) && c = apply\ c\ v \\
& evalMach\ (Add\ x\ y) && c = evalMach\ x\ (AddL\ c\ y)
\end{aligned}
$$

That is, evaluating a value calls the *apply* function with the current context and the integer value. Evaluating an addition evaluates the first argument and stores the second with the current context using the *AddL* constructor. Moving the lambda-abstracted terms to the left and applying the specification in the *AddL* case gives the following revised definition for *apply*:

$$
\begin{aligned}
& apply && :: Cont \rightarrow Value \rightarrow Value \\
& apply\ Top && v = v \\
& apply\ (AddR\ m\ c)\ n && = apply\ c\ (m + n) \\
& apply\ (AddL\ c\ y) && m = evalMach\ y\ (AddR\ m\ c)
\end{aligned}
$$

The *apply* function takes a context and a value and returns the value if the context is *Top*. When the context is *AddR* this represents the case when both sides of the addition have been evaluated, so the results are added together and the current

context is applied to the result. The *AddL* context represents evaluating the second argument to the addition, so the *evalMach* function is called and the result from the first argument and the current context is saved using the *AddR* context.

The original semantics can be recovered by passing in the equivalent of the initial identity continuation, the *Top* constructor:

$$eval\ e = evalMach\ e\ Top$$

In conclusion, we have now calculated a first-order, tail-recursive evaluator, i.e. an abstract machine.

Stack machine

It is relevant at this point to note that the above derivation is just one possible approach to producing an abstract machine. Different abstract machines may be generated by applying different program transformations. For example, as an alternative to the above machine, the defunctionalized continuation could be represented more conventionally as a stack of operations. Such operations consist of expressions still to be evaluated and values to be saved:

$$\begin{aligned}
&\textbf{type}\ Stack = [Op] \\
&\textbf{data}\ Op \quad = EVAL\ Expr \mid ADD\ Value
\end{aligned}$$

The *Cont* datatype in the previous machine is isomorphic to *Stack*, via the following conversion functions between the two types:

$$\begin{aligned}
&s2c && :: Stack \to Cont \\
&s2c\ [\,] && = Top \\
&s2c\ (EVAL\ e : s) && = AddL\ (s2c\ s)\ e \\
&s2c\ (ADD\ v : s) && = AddR\ v\ (s2c\ s)
\end{aligned}$$

$$\begin{aligned}
&c2s && :: Cont \to Stack \\
&c2s\ Top && = [\,] \\
&c2s\ (AddL\ c\ e) && = EVAL\ e : c2s\ c \\
&c2s\ (AddR\ v\ c) && = ADD\ v : c2s\ c
\end{aligned}$$

Using this stack type, the abstract machine that was derived in the previous section can be rewritten as follows, in which the two arguments have been packaged up as a pair that forms the configuration of the machine, and the function *apply* has been renamed to *exec* to capture the idea of executing the stack:

$$
\begin{array}{lll}
\mathit{evalMach} & & :: (\mathit{Expr}, \mathit{Stack}) \rightarrow \mathit{Value} \\
\mathit{evalMach}\ (\mathit{Val}\ v, & s) & = \mathit{exec}\ (s, v) \\
\mathit{evalMach}\ (\mathit{Add}\ x\ y, & s) & = \mathit{evalMach}\ (x, \mathit{EVAL}\ y : c) \\
\mathit{exec} & & :: (\mathit{Stack}, \mathit{Value}) \rightarrow \mathit{Value} \\
\mathit{exec} \quad ([\,], & v) & = v \\
\mathit{exec} \quad (\mathit{EVAL}\ y : c, & m) & = \mathit{evalMach}\ (y, \mathit{ADD}\ m : c) \\
\mathit{exec} \quad (\mathit{ADD}\ m : c, & n) & = \mathit{exec}\ (c, m + n)
\end{array}
$$

Formally, this new machine can itself be derived by following the same process as previously, but utilising the stack type instead of the context type during the defunctionalization step. Alternatively, it can be calculated from the abstract machine that was derived in the last section using the specification $\mathit{exec}\ (e, s) = \mathit{apply}\ e\ (\mathit{s2c}\ s)$, and similarly for the function *evalMach*.

3.4 Extended language

Now that the derivation process has been illustrated using a simple language of integers and addition, we extend this language with extra features to facilitate the study of the space and time behaviour of some programming examples. In a similar manner to the previous chapter, we extend the language with the untyped lambda calculus (variables, abstraction and application), together with lists. Instead of a general recursion operator, such as μ in the previous chapter, we introduce recursion over lists in the form of fold-right, as discussed in section 2.12.

The language is represented as the following Haskell datatype:

```
data Expr = Val Value | Add Expr Expr
          | Var Int | Abs Int Expr | App Expr Expr
```

$$\mid \mathit{Nil} \mid \mathit{Cons}\ \mathit{Expr}\ \mathit{Expr}$$
$$\mid \mathit{Foldr}\ \mathit{Expr}\ \mathit{Expr}\ \mathit{Expr}$$

That is, an expression is either a value, an addition, a variable, a lambda abstraction, an application, the empty list, a non-empty list, or a fold-right over an expression (where the arguments that replace each *Nil* and *Cons* are themselves represented as expressions). The structure of values will be considered shortly.

While this language doesn't allow us to define every function on lists, the fold operator is sufficient to define a large and interesting class of functions [40]. Extending this language with other datatypes, such as trees, would be straightforward, but for the purposes of this thesis we focus our attention on programs that process lists.

Evaluator

Prior to defining an evaluator for the extended language, we start by considering the lambda calculus subset. The denotational semantics for the lambda calculus presented in section 2.2 can be implemented directly by first defining expressible values as a datatype *ExprVal*, given by functions from *ExprVal* to *ExprVal*:

$$\textbf{newtype}\ \mathit{ExprVal} = \mathit{Fun}\ (\mathit{ExprVal} \to \mathit{ExprVal})$$

The semantics uses an environment to store values for variables, and for simplicity this is implemented as a list of (variable, value) pairs:

$$\textbf{type}\ \mathit{EnvDen} = [(\mathit{Int}, \mathit{ExprVal})]$$

Using these datatypes, evaluation can be defined as follows:

$$\begin{aligned}
&\mathit{den} && :: \mathit{Expr} \to \mathit{EnvDen} \to \mathit{ExprVal}\\
&\mathit{den}\ (\mathit{Var}\ x)\ \ \mathit{env} &&= \mathit{fromJust}\ (\mathit{lookup}\ x\ \mathit{env})\\
&\mathit{den}\ (\mathit{Abs}\ x\ e)\ \mathit{env} &&= \mathit{Fun}\ (\lambda v \to \mathit{den}\ e\ ((x, v) : \mathit{env}))\\
&\mathit{den}\ (\mathit{App}\ f\ e)\ \mathit{env} &&= \textbf{let}\ \mathit{Fun}\ f' = \mathit{den}\ f\ \mathit{env}\\
&&&\qquad\quad v\ \ \ \ = \mathit{den}\ e\ \mathit{env}\\
&&&\ \ \textbf{in}\ f'\ v
\end{aligned}$$

That is, a variable is evaluated by returning the value the variable is bound to in the environment. In turn, evaluation of an abstraction $Abs\ x\ e$ is performed by creating a function that takes a value v and evaluates e in the environment extended with x bound to v. Lastly, an application $App\ f\ e$ is evaluated by first evaluating f, then evaluating e, and finally applying the result of f to the result of e.

The current representation of values, however, is impractical. For example, at present we cannot print the result of an evaluation, because it is a function. To obtain a more convenient value representation the function space $ExprVal \to ExprVal$ can be removed by *closure conversion*. This is a simple example of defunctionalization, by first identifying that a function $(\lambda v \to den\ e\ ((x, v) : env))$ is created in the Abs case, and then defining a datatype with arguments for the free variables x, e and env. The resulting datatype is a closure, where an abstraction is paired with an environment, though we represent this in a flattened form by the following datatype:

$$\textbf{data}\ Value = Clo\ Int\ Expr\ Env$$

An abstraction can be recovered from a closure by repeatedly substituting in the value for each variable in the environment into the expression part of the closure. The original representation of a value is then replaced by the new datatype and the associated application function is inlined, to give:

$$\begin{aligned}
&\textbf{type}\ Env && = [(Int, Value)] \\
&eval && :: Expr \to Env \to Value \\
&eval\ (Var\ x)\ env && = fromJust\ (lookup\ x\ env) \\
&eval\ (Abs\ x\ e)\ env && = Clo\ x\ e\ env \\
&eval\ (App\ f\ e)\ env && = \textbf{let}\ (Clo\ x\ e'\ env') = eval\ f\ env \\
& && \qquad\quad v = eval\ e\ env \\
& && \quad \textbf{in}\ eval\ e'\ ((x, v) : env')
\end{aligned}$$

We can now define a value in the extended language as either an integer, a closure, or a list containing further values:

$$\begin{aligned}
\textbf{data}\ Value = {} & Const\ Int \mid Clo\ Int\ Expr\ Env \\
& \mid VCons\ Value\ Value \mid VNil
\end{aligned}$$

Note that there are two representations of lists in the language, one containing elements of type *Expr* and the other only containing evaluated expressions, of type *Value*. The reason for this is to ensure that there is no need to repeatedly iterate over a list to check if each element is fully evaluated which would, particularly in the call-by-value strategy, introduce an artificial evaluation overhead.

The evaluator for the remainder of the language is given below:

$$
\begin{aligned}
&\mathit{eval} && :: \quad \mathit{Expr} \to \mathit{Env} \to \mathit{Value} \\
&\mathit{eval}\ (\mathit{Val}\ v) && \mathit{env} = v \\
&\mathit{eval}\ (\mathit{Add}\ x\ y) && \mathit{env} = \mathbf{let}\ \mathit{Const}\ m = \mathit{eval}\ x\ \mathit{env} \\
&&& \qquad\qquad \mathit{Const}\ n = \mathit{eval}\ y\ \mathit{env} \\
&&& \quad\ \mathbf{in}\ \mathit{Const}\ (m + n) \\
&\mathit{eval}\ \mathit{Nil} && \mathit{env} = \mathit{VNil} \\
&\mathit{eval}\ (\mathit{Cons}\ x\ \mathit{xs})\ \mathit{env} && = \mathit{VCons}\ (\mathit{eval}\ x\ \mathit{env})\ (\mathit{eval}\ \mathit{xs}\ \mathit{env}) \\
&\mathit{eval}\ (\mathit{Foldr}\ f\ v\ \mathit{xs})\ \mathit{env} && = \mathbf{case}\ \mathit{eval}\ \mathit{xs}\ \mathit{env}\ \mathbf{of} \\
&\quad \mathit{VNil} \to \mathit{eval}\ v\ \mathit{env} \\
&\quad \mathit{VCons}\ y\ \mathit{ys} \to && \mathbf{let}\ f' = \mathit{eval}\ f\ \mathit{env} \\
&&& \qquad\quad z = \mathit{eval}\ (\mathit{Foldr}\ (\mathit{Val}\ f')\ v\ (\mathit{Val}\ \mathit{ys}))\ \mathit{env} \\
&&& \quad\ \mathbf{in}\ \mathit{eval}\ (\mathit{App}\ (\mathit{App}\ (\mathit{Val}\ f')\ (\mathit{Val}\ y))\ (\mathit{Val}\ z))\ [\,]
\end{aligned}
$$

That is, values are already evaluated, so are simply returned. Addition is performed as in the original evaluator, by first evaluating both sides to an integer and then adding them together. Evaluating a *Cons* consists of evaluating the first and second arguments (the head and the tail of the list) and then re-assembling them using the *VCons* constructor to make an evaluated list. Evaluation of the *Foldr* case proceeds by first evaluating the list argument and then performing a case analysis on the resulting list. If the list is empty, then the result is the evaluation of the second argument, v. In the non-empty case, the function argument is first evaluated, then the fold-right applied to the tail of the list, and finally the function is applied to the head of the list and the result of folding the tail of the list. This evaluation is performed with an empty environment, because all the expressions are values at that point, and hence do not contain free variables. Note that in the final line of the *Foldr*

case, it would have been possible to inline the definition of *eval*, but we prefer to keep a clean separation between the meaning of each language construct in our evaluator.

The evaluation of *Foldr* could have been specified in different ways. The completely call-by-value approach would be to evaluate the arguments in left-to-right order, with the first two arguments being evaluated before the list argument. However, for the case when the list is empty, the function argument to *Foldr* would be evaluated even though it is not required. The approach taken in our evaluator is to evaluate the list argument first, to allow pattern matching, and then evaluate the other arguments depending on the nature of the resulting list. In particular, when the list is empty, only the second argument to *Foldr* is evaluated. This was a design decision, justified by the fact that the purely call-by-value approach would introduce some artificial behaviour of the *Foldr* function in that, when the list supplied is empty, the function argument would still be evaluated. When the lambda calculus is extended with a conditional function, for example, it is not implemented to expand both branches under call-by-value evaluation, but more efficiently by evaluating the condition first and then one branch depending on the value of the condition. In practice, this has little effect because the function supplied to the *Foldr* is often an abstraction and therefore is already a value.

Abstract machine

Performing the same calculational process as in section 3.3, that is, conversion into continuation-passing style to make the evaluation order explicit, followed by defunctionalization to make the evaluator first-order, results in the abstract machine for the extended language given below. We omit the details of the actual calculation process itself, but we will return in a later section of the thesis to discuss how this process may be automated, or mechanically verified.

$$\begin{aligned}\textbf{data}\ \mathit{Cont} = {} & \mathit{Top} \mid \mathit{AddL}\ \mathit{Cont}\ \mathit{Expr}\ \mathit{Env} \mid \mathit{AddR}\ \mathit{Value}\ \mathit{Cont} \\ & \mid \mathit{AppL}\ \mathit{Cont}\ \mathit{Expr}\ \mathit{Env} \mid \mathit{AppR}\ \mathit{Value}\ \mathit{Cont} \\ & \mid \mathit{Fold1}\ \mathit{Expr}\ \mathit{Expr}\ \mathit{Env}\ \mathit{Cont} \mid \mathit{Fold2}\ \mathit{Cont}\ \mathit{Expr}\ \mathit{Env}\ \mathit{Value}\ \mathit{Value}\end{aligned}$$

```
    | Fold3 Cont Value Value
    | ConsL Cont Expr Env | ConsR Value Cont

evalMach                    :: Expr → Env → Cont → Value
evalMach (Val v)       env  c = apply c v
evalMach (Add x y)     env  c = evalMach x env (AddL c y env)
evalMach (Var x)       env  c = apply c (fromJust (lookup x env))
evalMach (Abs x e)     env  c = apply c (Clo x e env)
evalMach (App f e)     env  c = evalMach f env (AppL c e env)
evalMach (Nil)         env  c = apply c VNil
evalMach (Cons x xs) env  c = evalMach x env (ConsL c xs env)
evalMach (Foldr f v xs) env c = evalMach xs env (Fold1 f v env c)

apply                                  :: Cont → Value → Value
apply Top                   v          = v
apply (AddL c y env)        m          = evalMach y env (AddR m c)
apply (AddR (Const m) c) (Const n) = apply c (Const (m + n))
apply (AppL c e env)        v'         = evalMach e env (AppR v' c)
apply (AppR v' c)           v          = let (Clo x e' env') = v'
                                         in evalMach e' ((x, v) : env') c
apply (Fold1 f v env c)     ys         = case ys of
  VNil → evalMach v env c
  VCons z zs → evalMach f env (Fold2 c v env z zs)
apply (Fold2 c v env z zs) f'          =
  evalMach (Foldr (Val f') v (Val zs)) env (Fold3 c f' z)
apply (Fold3 c f' z)        x          =
  evalMach (App (App (Val f') (Val z)) (Val x)) [] c
apply (ConsL c xs env)      v          = evalMach xs env (ConsR v c)
apply (ConsR v c)           vs         = apply c (VCons v vs)
```

The details of how the abstract machine works need not concern us, because we do not want to reason about programs at this low-level. All that is important is that the execution behaviour comes directly from the semantics of the original evaluator,

and at this level all execution steps and data structures are explicit.

3.5 Summary

This chapter has reviewed the concept of abstract machines and, starting with an evaluator for a simple language, has shown how it may be transformed into an abstract machine by a process of systematic calculation. The language was then extended with extra features that will facilitate the study of the space and time behaviour of some familiar functions. The next two chapters will show how to instrument this derived machine with time and space information. Once we have an instrumented machine, we can then apply the same calculational techniques, but in the reverse order, with the result being high-level functions that measure space and time usage. In this manner, we can reason about properties of functions as high-level evaluators, rather than low-level machines. These functions will then be used to look at the space and time performance of some example functions expressed in the extended language.

Chapter 4

Adding time

Traditionally, when reasoning about the time usage of functional programs, an abstract notion of evaluation step is adopted, rather than using real time costs. As discussed in section 1.6, the notion of evaluation step that is conventionally adopted is β-reduction. Here, however, we propose that a more realistic measure is to instead count transitions in an underlying abstract machine. The work in this chapter has been published in [49], but is presented here in more detail.

In the previous chapter we showed how an abstract machine may be calculated from an evaluator for a language, and here we consider how to instrument such a machine with time information. The time requirements are abstracted by taking the number of transitions required by the machine. We first consider the simple language of integers and addition, using the machine derived in section 3.3. We then show how the result generalises to the machine for our extended language, and how the derived semantics can be used to reason about the time performance of a number of familiar functions. Finally, we put these results into context, by presenting benchmarks using both the Hugs system and the Glasgow Haskell Compiler.

4.1 Measuring time

First of all, we define the time required to evaluate an expression as the number of transitions of the underlying abstract machine. Consider a reduction sequence that

starts with the state $x_0 = (e, Top)$ and ends with the value v:

$$x_0 \to x_1 \to \cdots \to x_n \to v$$

The number of steps required for this reduction, written *steps* e, is therefore the length of the list $[x_0, x_1, \cdots x_n, v]$, which is equal to $n + 1$. Calculating the length of a list can be defined by recursion over the structure of the list, returning zero if the list is empty and, in the non-empty case, one plus the length of the tail:

$$\begin{aligned} &\mathit{length}\ [\,] &&= 0 \\ &\mathit{length}\ (x : xs) &&= 1 + \mathit{length}\ xs \end{aligned}$$

However, if we incorporate this method of measuring the number of the transitions into the abstract machine the result will no longer be tail-recursive, because we will have an addition still to perform when the recursive call returns. Alternatively we can define the length of a list by using an accumulator:

$$\begin{aligned} &\mathit{length} &&= \mathit{length}'\ 0 \\ &\mathit{length}'\ a\ [\,] &&= a \\ &\mathit{length}'\ a\ (x : xs) &&= \mathit{length}\ (a + 1)\ xs \end{aligned}$$

The accumulator is initialised to zero and simply returned in the empty list case, and incremented by one in the non-empty case. This definition is tail-recursive, and in this way the number of transitions can be counted whilst still ensuring that the abstract machine itself is tail-recursive.

Step counting machine

The abstract machine derived in section 3.3 can be modified to count the number of transitions by simply adding a step counter that is incremented each time a transition, in the form of a function call to *evalMach* or *apply*, is made:

$$\begin{aligned} &\textbf{type}\ \mathit{Step} &&= \mathit{Int} \\ &\mathit{stepMach} &&:: (\mathit{Expr}, \mathit{Step}) \to \mathit{Cont} \to (\mathit{Value}, \mathit{Step}) \\ &\mathit{stepMach}\ (\mathit{Val}\ v, s)\quad c &&= \mathit{apply}'\ c\ (v, s + 1) \end{aligned}$$

$$
\begin{array}{lll}
\mathit{stepMach}\ (\mathit{Add}\ x\ y, s)\ c & = \mathit{stepMach}\ (x, s+1)\ (\mathit{AddL}\ c\ y) \\
\mathit{apply}' & :: \mathit{Cont} \to (\mathit{Value}, \mathit{Step}) \to (\mathit{Value}, \mathit{Step}) \\
\mathit{apply}'\ \mathit{Top} & (v, s) = (v, s+1) \\
\mathit{apply}'\ (\mathit{AddL}\ c\ y) & (m, s) = \mathit{stepMach}\ (y, s+1)\ (\mathit{AddR}\ m\ c) \\
\mathit{apply}'\ (\mathit{AddR}\ m\ c) & (n, s) = \mathit{apply}'\ c\ (m+n, s+1)
\end{array}
$$

In this case we are only counting the machine transitions, and the actual addition of the integers $m + n$ that occurs in the *AddR* case of the *apply′* function is defined to happen instantly. The machine could be extended with an additional factor that represents the number of steps to perform an addition if so desired.

In the above machine an explicit approach to instrumentation has been taken. An alternative would be to adopt a monadic approach to threading around the necessary extra information. For example, we could use the tick monad [50] to add time information to the machine. However, while this would have the advantage of giving a more structured machine, reasoning about monadic programs typically proceeds by first in-lining the definitions of the monadic operators [51], which means that for our purposes we would gain little benefit from a monadic approach. For this reason, we prefer the explicit approach to instrumentation.

An expression can now be evaluated by applying the *stepMach* function with the step count initialised to zero and the context initialised to *Top*. The value of the expression is the first component of the resulting pair, while the number of steps is the second component, as captured by the following definitions:

$$
\begin{array}{l}
\mathit{eval}\ e\ = \mathit{fst}\ (\mathit{stepMach}\ (e, 0)\ \mathit{Top}) \\
\mathit{steps}\ e = \mathit{snd}\ (\mathit{stepMach}\ (e, 0)\ \mathit{Top})
\end{array}
$$

Now, the same program transformations as in the previous chapter can be performed, but in the reverse order, to calculate a high-level function that counts steps corresponding to the number of transitions of the abstract machine.

Step counting tail-recursive evaluator

The first stage in the reverse process is to refunctionalize the representation of the continuation. The original continuation was a function of type $Value \rightarrow Value$; the new continuation also includes a step count, and has the following type:

$$\textbf{type } Con' = (Value, Step) \rightarrow (Value, Step)$$

The definition of the refunctionalized function *stepTail* can be calculated by induction on the expression argument e, starting from the following specification:

$$stepTail\ (e, s)\ (apply'\ c) = stepMach\ (e, s)\ c$$

This equation states that *stepTail* gives the same result as *stepMach*, except that it takes a continuation as its second argument, instead of a context.

Case : $e = Val\ v$

$$\begin{array}{ll} & stepTail\ (Val\ v, s)\ (apply'\ c) \\ = & \quad \{\ \text{specification}\ \} \\ & stepMach\ (Val\ v, s)\ c \\ = & \quad \{\ \text{definition of } stepMach\ \} \\ & apply'\ c\ (v, s+1) \end{array}$$

Case : $e = Add\ x\ y$

$$\begin{array}{ll} & stepTail\ (Add\ x\ y, s)\ (apply'\ c) \\ = & \quad \{\ \text{specification}\ \} \\ & stepMach\ (Add\ x\ y, s)\ c \\ = & \quad \{\ \text{definition of } stepMach\ \} \\ & stepMach\ (x, s+1)\ (AddL\ c\ y) \\ = & \quad \{\ \text{inductive assumption for } x\ \} \\ & stepTail\ (x, s+1)\ (apply'\ (AddL\ c\ y)) \\ = & \quad \{\ \text{definition of } apply'\ \} \\ & stepTail\ (x, s+1)\ (\lambda(m, s') \rightarrow stepMach\ (y, s'+1)\ (AddR\ m\ c)) \end{array}$$

$$
\begin{aligned}
= \quad & \{ \text{ inductive assumption for } y \ \} \\
& \mathit{stepTail}\ (x, s+1)\ (\lambda(m, s') \rightarrow \\
& \mathit{stepTail}\ (y, s'+1)\ (\mathit{apply}'\ (\mathit{AddR}\ m\ c))) \\
= \quad & \{ \text{ definition of } \mathit{apply}' \ \} \\
& \mathit{stepTail}\ (x, s+1)\ (\lambda(m, s') \rightarrow \\
& \mathit{stepTail}\ (y, s'+1)\ (\lambda(n, s'') \rightarrow \mathit{apply}'\ c\ (m+n, s''+1)))
\end{aligned}
$$

The next step is to generalise the expression $\mathit{apply}'\ c$ to an arbitrary continuation c, which gives the following refunctionalized version:

$$
\begin{aligned}
& \mathit{stepTail} && :: (\mathit{Expr}, \mathit{Step}) \rightarrow \mathit{Con}' \rightarrow (\mathit{Value}, \mathit{Step}) \\
& \mathit{stepTail}\ (\mathit{Val}\ v) && s\ c = c\ (v, s+1) \\
& \mathit{stepTail}\ (\mathit{Add}\ x\ y)\ s\ c = \ & \mathit{stepTail}\ x\ (s+1)\ (\lambda(m, s') \rightarrow \\
& && \mathit{stepTail}\ y\ (s'+1)\ (\lambda(n, s'') \rightarrow \\
& && c\ (m+n, s''+1)))
\end{aligned}
$$

The step counting semantics can now be redefined as follows:

$$\mathit{steps}\ e = \mathit{snd}\ (\mathit{stepTail}\ (e, 0)\ (\lambda(v, s) \rightarrow (v, s+1)))$$

Note that the initial continuation function increments the step count by one, which corresponds to the transition in the *Top* case of the apply' function. The next step in the derivation process is to remove the continuation.

Step counting evaluator with accumulator

The step counting evaluator can be transformed from continuation-passing style back to a direct style by starting with the following specification:

$$c\ (\mathit{stepAcc}\ (e, s)) = \mathit{stepTail}\ (e, s)\ c$$

This equation expresses that the continuation applied to the new function *stepAcc* gives the same final result as supplying the continuation as an argument to *stepTail*. The definition for *stepAcc* can now be calculated by induction on e:

Case : $e = \mathit{Val}\ n$

$$
\begin{array}{ll}
 & c\ (stepAcc\ (Val\ n, s)) \\
= & \quad \{\ \text{specification}\ \} \\
 & stepTail\ (Val\ n, s)\ c \\
= & \quad \{\ \text{definition of } stepTail\ \} \\
 & c\ (v, s+1)
\end{array}
$$

Case : $e = Add\ x\ y$

$$
\begin{array}{ll}
 & c\ (stepAcc\ (Add\ x\ y, s)) \\
= & \quad \{\ \text{specification}\ \} \\
 & stepTail\ (Add\ x\ y, s)\ c \\
= & \quad \{\ \text{definition of } stepTail\ \} \\
 & stepTail\ x\ (s+1)\ (\lambda(m, s') \rightarrow \\
 & stepTail\ y\ (s'+1)\ (\lambda(n, s'') \rightarrow c\ (m+n, s''+1))) \\
= & \quad \{\ \text{inductive assumption for } x\ \} \\
 & (\lambda(m, s') \rightarrow stepTail\ y\ (s'+1)\ (\lambda(n, s'') \\
 & \rightarrow c\ (m+n, s''+1)))\ stepAcc\ (x, s+1) \\
= & \quad \{\ \text{introduce let-binding}\ \} \\
 & \textbf{let}\ (m, s') = stepAcc\ (x, s+1) \\
 & \textbf{in}\ stepTail\ y\ (s'+1)\ (\lambda(n, s'') \rightarrow c\ (m+n, s''+1)) \\
= & \quad \{\ \text{inductive assumption for } y\ \} \\
 & \textbf{let}\ (m, s') = stepAcc\ (x, s+1) \\
 & \textbf{in}\ (\lambda(n, s'') \rightarrow c\ (m+n, s''+1))\ stepAcc\ (y, s'+1) \\
= & \quad \{\ \text{introduce let-binding}\ \} \\
 & \textbf{let}\ (m, s') = stepAcc\ (x, s+1) \\
 & \quad\ (n, s'') = stepAcc\ (y, s'+1) \\
 & \textbf{in}\ c\ (m+n, s''+1)
\end{array}
$$

The resulting evaluator is:

$$\begin{aligned}
&\mathit{stepAcc} && :: (\mathit{Expr}, \mathit{Step}) \to (\mathit{Value}, \mathit{Step}) \\
&\mathit{stepAcc}\ (\mathit{Val}\ v) && s = (v, s+1) \\
&\mathit{stepAcc}\ (\mathit{Add}\ x\ y)\ s = \mathbf{let}\ && (m, s') = \mathit{stepAcc}\ (x, s+1) \\
& && (n, s'') = \mathit{stepAcc}\ (y, s'+1) \\
& && \mathbf{in}\ (m+n, s''+1)
\end{aligned}$$

The new step counting function becomes:

$$\mathit{steps}\ e = \mathit{snd}\ (\mathit{stepAcc}\ (e, 0)) + 1$$

At the moment, the step count is threaded through the function as an accumulator. However, the need for an accumulator can also be removed by calculation.

Step counting evaluator

The new function can be calculated using the following specification:

$$\mathit{stepAcc}\ (e, i) = \mathbf{let}\ (v, r) = \mathit{stepEval}\ e\ \mathbf{in}\ (v, r+i)$$

This equation states that the new non-accumulator version, *stepEval*, produces the same value (first part of the pair), and that the step count (second part of the pair) is equal to the accumulator step count plus the initial step count. As usual, the definition of the new function can be calculated by induction on e.

Case : $e = \mathit{Val}\ n$

$$\begin{aligned}
& \mathbf{let}\ (v, r) = \mathit{stepEval}\ (\mathit{Val}\ v)\ \mathbf{in}\ (v, r+i) \\
= & \quad \{\ \text{specification}\ \} \\
& \mathit{stepAcc}\ (\mathit{Val}\ n, i) \\
= & \quad \{\ \text{definition of } \mathit{stepAcc}\ \} \\
& (n, 1+i)
\end{aligned}$$

Case : $e = \mathit{Add}\ x\ y$

$$
\begin{aligned}
& \textbf{let } (v, r) = \mathit{stepEval}\ (\mathit{Add}\ x\ y) \textbf{ in } (v, r + i) \\
= & \quad \{\ \text{specification}\ \} \\
& \mathit{stepAcc}\ (\mathit{Add}\ x\ y, i) \\
= & \quad \{\ \text{definition of } \mathit{stepAcc}\ \} \\
& \textbf{let } (m, s') = \mathit{stepAcc}\ (x, i + 1) \\
& \quad\ (n, s'') = \mathit{stepAcc}\ (y, s' + 1) \\
& \textbf{in } (m + n, s'' + 1) \\
= & \quad \{\ \text{inductive assumptions for } x \text{ and } y\ \} \\
& \textbf{let } (m, s') = \textbf{let } (xm, xs) = \mathit{stepEval}\ x \textbf{ in } (xm, xs + (i + 1)) \\
& \quad\ (n, s'') = \textbf{let } (ym, ys) = \mathit{stepEval}\ y \textbf{ in } (ym, ys + (s' + 1)) \\
& \textbf{in } (m + n, s'' + 1) \\
= & \quad \{\ \text{substitute } xm = m,\ s' = xs + (s + 1),\ ym = n,\ s'' = ys + (s' + 1)\ \} \\
& \textbf{let } (m, s') = \textbf{let } (xm, xs) = \mathit{stepEval}\ x \textbf{ in } (xm, xs + (i + 1)) \\
& \quad\ (n, s'') = \textbf{let } (ym, ys) = \mathit{stepEval}\ y \textbf{ in } (ym, ys + (s' + 1)) \\
& \textbf{in } (m + n, (ys + ((xs + (i + 1)) + 1)) + 1) \\
= & \quad \{\ \text{rename and apply arithmetic}\ \} \\
& \textbf{let } (m, s) = \mathit{stepEval}\ x \\
& \quad\ (n, s') = \mathit{stepEval}\ y \\
& \textbf{in } (m + n, s' + s + 3 + i)
\end{aligned}
$$

Simplifying each side, by removing the $+i$, yields the following definition:

$$
\begin{aligned}
& \mathit{stepEval} && :: \mathit{Expr} \to (\mathit{Value}, \mathit{Step}) \\
& \mathit{stepEval}\ (\mathit{Val}\ v) && = (v, 1) \\
& \mathit{stepEval}\ (\mathit{Add}\ x\ y) && = \textbf{let } (m, s) = \mathit{stepEval}\ x \\
& && \qquad\ (n, s') = \mathit{stepEval}\ y \\
& && \quad \textbf{in } (m + n, s' + s + 3)
\end{aligned}
$$

In turn, the step counting function now becomes:

$$\mathit{steps}\ e = \mathit{snd}\ (\mathit{stepEval}\ e) + 1$$

The final stage is to calculate a standalone *steps* function. This function will take an expression and return the number of steps required to evaluate the expression, calling

the original evaluator when the result of evaluation is required.

Step counting function

The direct step counting function can be produced by routine calculation starting from the equation at the end of the previous section:

$$
\begin{aligned}
&\mathit{steps}\ e && :: \mathit{Expr} \to \mathit{Step} \\
&\mathit{steps}\ e && = \mathit{steps}'\ e + 1 \\
&\mathit{steps}'\ (\mathit{Val}\ v) && = 1 \\
&\mathit{steps}'\ (\mathit{Add}\ x\ y) && = \mathit{steps}'\ x + \mathit{steps}'\ y + 3
\end{aligned}
$$

The derived *steps* function shows that the number of steps required to evaluate an expression is given by an auxiliary function steps' plus a constant one, which arises from the transition in the abstract machine that evaluates the initial (Top) context. In steps', the number of steps to evaluate an integer is a constant one, and the number of steps to evaluate an addition is the number of steps to evaluate each argument plus a constant three. In this manner, we now see the precise overhead of each addition: if we were counting β-reductions then this would only have been a single step of evaluation, whereas we now see that each addition takes three steps of evaluation in the underlying machine. While in isolation the difference between one step and three steps may not be particularly important, when many additions are performed (for example, if the expression being evaluated is large) the cumulative effect of such extra steps may be considerable. Note that for this simple language calling the original evaluator in the derived step function is not required.

4.2 Time function

Performing the derivation process for the extended language presented in section 3.4, which as previously proceeds by calculation, yields the following *steps* function:

$$
\begin{aligned}
&\mathit{steps} && :: \mathit{Expr} \to \mathit{Step} \\
&\mathit{steps}\ e && = \mathit{steps}'\ e\ [\,] + 1
\end{aligned}
$$

$$
\begin{array}{ll}
steps'\ (Val\ v) & env = 1 \\
steps'\ (Var\ x) & env = 1 \\
steps'\ (Abs\ x\ e) & env = 1 \\
steps'\ (App\ f\ e) & env = \textbf{let}\ (Clo\ x\ e'\ env') = eval\ f\ env \\
 & \qquad\qquad v = eval\ e\ env \\
 & \qquad\quad \textbf{in}\ steps'\ f\ env + steps'\ e\ env + \\
 & \qquad\qquad steps'\ e'\ ((x, v) : env') + 3 \\
steps'\ (Add\ x\ y) & env = steps'\ x\ env + steps'\ y\ env + 3 \\
steps'\ (Nil) & env = 1 \\
steps'\ (Cons\ x\ xs) & env = steps'\ x\ env + steps'\ xs\ env + 3 \\
steps'\ (Foldr\ f\ v\ xs) & env = steps'\ xs\ env + \textbf{case}\ eval\ xs\ env\ \textbf{of} \\
\quad VNil \rightarrow steps'\ v\ env + 2 & \\
\quad VCons\ y\ ys \rightarrow \textbf{let}\ f' = eval\ f\ env & \\
 & \qquad\qquad x = eval\ (Foldr\ (Val\ f')\ v\ (Val\ ys))\ env \\
 & \qquad\quad \textbf{in}\ steps'\ f\ env + steps'\ (Foldr\ (Val\ f')\ v\ (Val\ ys))\ env + \\
 & \qquad\quad steps'\ (App\ (App\ (Val\ f')\ (Val\ y))\ (Val\ x))\ [\,] + 4
\end{array}
$$

As mentioned earlier, the derived function calls the original evaluator when the result of evaluation is required. For example, in the *Foldr* case a case analysis has to be performed on the evaluated third argument to know if it is empty or not, and hence whether to supply the v argument or to keep folding.

4.3 Examples

We can now use the derived *steps* function to analyse the time performance of some programming examples. Each step counting function produced was simplified by hand, and then QuickCheck [52] was used to verify that the resulting function was correct with respect to the behaviour of the abstract machine, to check that no errors had been introduced during simplification.

Sum

Summing a list of integers can be expressed using fold-right:

$$\begin{array}{lcl} sum\ [\,] & = & 0 \\ sum\ (x : xs) & = & x + sum\ xs \end{array} \quad \Leftrightarrow \quad sum\ xs = foldr\ (+)\ 0\ xs$$

The fold-right definition replaces each list constructor : with +, and the [] at the end of the list with 0, the unit of addition. First we translate the definition of *sum* into the language syntax, and then call the *steps* function on the application of *sum* to an argument. We can expand the expression $steps\ (App\ sum\ (Val\ xs))$ using the definition of *steps* and then simplify, to give the following recursive cost equation:

$$\begin{array}{l} steps\ (App\ sum\ (Val\ xs)) = \textbf{case}\ xs\ \textbf{of} \\ \quad VNil \rightarrow 10 \\ \quad VCons\ y\ ys \rightarrow 21 + steps\ (App\ sum\ (Val\ ys)) \end{array}$$

In order to express this equation in a closed form, we first define the length function over *Value* lists:

$$\begin{array}{ll} lengthVal & :: Value \rightarrow Int \\ lengthVal\ VNil & = 0 \\ lengthVal\ (VCons\ y\ ys) & = 1 + lengthVal\ ys \end{array}$$

The cost of evaluating the *sum* function can then be expressed in terms of $lengthVal$, to give the following non-recursive equation:

$$steps\ (App\ sum\ (Val\ xs)) = 10 + 21 * (lengthVal\ xs)$$

The equation expresses that the number of steps required to sum a list is linear in the length of the list. More importantly, however, the equation makes precise the constant factors involved, namely that each number in the list incurs 21 steps of evaluation, and that an additional overhead of 10 steps is required. We will return to the issue of how such results compare to experimental practice when we consider benchmarking at the end of the chapter.

The recursive cost equation can be expressed in terms of *length* because the num-

ber of steps required to evaluate the addition of two values is a constant. In general, the cost of evaluating a function expressed using *Foldr* can be defined in closed-form as a fold-right over the *Value* list structure, by returning a pair consisting of the evaluated value and the step count.

Sum with an accumulator

An alternative definition of *sum* is to use an accumulator. In this case, a fold-right can be used to generate a function which is applied to the identity function in the empty list case, and in the non-empty case adds the head of the list to the accumulator:

$$\begin{array}{lcl} sumAcc\ [\,]\ a & = & a \\ sumAcc\ (x : xs)\ a & = & sumAcc\ xs\ (a + x) \end{array} \quad \Leftrightarrow \quad \begin{array}{l} sumAcc\ xs = foldr\ f\ id\ xs\ 0 \\ \quad \textbf{where}\ f\ x\ g\ a = g\ (a + x) \end{array}$$

Using an accumulator could potentially save on space, because additions could be performed without having to expand the whole list first. It would be useful to know what effect an accumulator has on the number of steps taken.

Translating the accumulator version and applying *steps* gives a result of the same form as previously, i.e. linear in the length of the list, but the constant values are larger, because there is an additional overhead in evaluating the extra abstractions:

$$steps\ (App\ sumAcc\ (Val\ xs)) = 15 + 26 * (lengthVal\ xs)$$

To quantify these results, the additional overhead of the *sumAcc* definition results in approximately 24% more steps than the original definition of *sum*.

The accumulator version is more traditionally written using fold-left, via the definition $suml\ xs\ a = foldl\ (+)\ a\ xs$. To compare the performance overhead of using this definition compared to the fold-right one, we now add fold-left to our language. The semantics of fold-left are discussed in detail in section 6.2, and we implement it in our call-by-value language in the following way. First of all, evaluating an expression of the form $Foldl\ f\ a\ xs$ consists of evaluating the list xs. If the list if *Nil* then the accumulator a is evaluated. Otherwise, the list is of the form $Cons\ y\ ys$, and we first evaluate the remaining arguments, then apply the function to the accumulator and

the list element y to obtain a new accumulator value, which is finally passed to fold the remainder of the list ys:

$$
\begin{aligned}
&\mathit{eval}\ (\mathit{Foldl}\ f\ a\ \mathit{xs})\ \mathit{env} = \mathbf{case}\ \mathit{eval}\ \mathit{xs}\ \mathit{env}\ \mathbf{of}\\
&\quad \mathit{VNil} \qquad\quad \rightarrow \mathit{eval}\ a\ \mathit{env}\\
&\quad \mathit{VCons}\ y\ \mathit{ys} \rightarrow \mathbf{let}\ f' = \mathit{eval}\ f\ \mathit{env}\\
&\qquad\qquad\qquad\qquad\quad a' = \mathit{eval}\ a\ \mathit{env}\\
&\qquad\qquad\qquad\qquad\quad a'' = \mathit{eval}\ (\mathit{App}\ (\mathit{App}\ (\mathit{Val}\ f')\ (\mathit{Val}\ a'))\ (\mathit{Val}\ y))\ [\,]\\
&\qquad\qquad\qquad\quad \mathbf{in}\ \mathit{eval}\ (\mathit{Foldl}\ (\mathit{Val}\ f')\ (\mathit{Val}\ a'')\ (\mathit{Val}\ \mathit{ys}))\ [\,]
\end{aligned}
$$

Applying the same calculational technique presented above, we can derive the following $steps'$ definition for the fold-left case:

$$
\begin{aligned}
&\mathit{steps}'\ (\mathit{Foldl}\ f\ a\ \mathit{xs})\ \mathit{env} = \mathit{steps}'\ a\ \mathit{env} + \mathit{steps}'\ \mathit{xs}\ \mathit{env} +\\
&\quad \mathbf{case}\ \mathit{eval}\ \mathit{xs}\ \mathit{env}\ \mathbf{of}\\
&\qquad \mathit{VNil} \qquad\quad \rightarrow 2\\
&\qquad \mathit{VCons}\ y\ \mathit{ys} \rightarrow \mathbf{let}\ f' = \mathit{eval}\ f\ \mathit{env}\\
&\qquad\qquad\qquad\qquad\quad a' = \mathit{eval}\ a\ \mathit{env}\\
&\qquad\qquad\qquad\qquad\quad a'' = \mathit{eval}\ (\mathit{App}\ (\mathit{App}\ (\mathit{Val}\ f')\ (\mathit{Val}\ a'))\ (\mathit{Val}\ y))\ [\,]\\
&\qquad\qquad\qquad\quad \mathbf{in}\ \mathit{steps}'\ f\ \mathit{env} +\\
&\qquad\qquad\qquad\qquad \mathit{steps}'\ (\mathit{App}\ (\mathit{App}\ (\mathit{Val}\ f')\ (\mathit{Val}\ a'))\ (\mathit{Val}\ y))\ [\,] +\\
&\qquad\qquad\qquad\qquad \mathit{steps}'\ (\mathit{Foldl}\ (\mathit{Val}\ f')\ (\mathit{Val}\ a'')\ (\mathit{Val}\ \mathit{ys}))\ [\,] + 5
\end{aligned}
$$

Applying the *steps* function to the fold-left definition of *sum* gives the following result:

$$\mathit{steps}\ (\mathit{App}\ \mathit{suml}\ (\mathit{Val}\ \mathit{xs})) = 10 + 23 * \mathit{length}\mathit{Val}\ \mathit{xs}$$

In conclusion, we see that the fold-left implementation still requires more steps than the original fold-right version, but has a lower overhead than the fold-right with an accumulator definition. In particular, the number of steps required for each element reduces from 26 to 23, and the cost for the empty list reduces from 15 to 10.

Reverse

Reversing a list can be expressed directly as a fold-right by appending the reversed tail of the list to a singleton list comprising the head:

$$\begin{array}{lcl} \mathit{reverse}\ [\,] & = & [\,] \\ \mathit{reverse}\ (x : \mathit{xs}) & = & \mathit{reverse}\ \mathit{xs} +\!\!+ [x] \end{array} \quad \Leftrightarrow \quad \begin{array}{l} \mathit{reverse}\ \mathit{xs} = \mathit{foldr}\ f\ [\,]\ \mathit{xs} \\ \quad \textbf{where}\ f\ x\ \mathit{xs} = \mathit{xs} +\!\!+ [x] \end{array}$$

In order to determine the time performance of *reverse*, we first need to analyse the *append* function, which can be expressed as a fold-right as follows:

$$\mathit{append}\ \mathit{xs}\ \mathit{ys} = \mathit{foldr}\ (:)\ \mathit{ys}\ \mathit{xs}$$

Translating the definition of *append* into our language syntax and applying the *steps* function gives the following number of steps to append two list arguments:

$$\mathit{steps}\ (\mathit{App}\ (\mathit{App}\ \mathit{append}\ (\mathit{Val}\ \mathit{xs}))\ (\mathit{Val}\ \mathit{ys})) = 15 + 21 * (\mathit{lengthVal}\ \mathit{xs})$$

Hence, the number of steps required to evaluate an *append* is proportional to the length of the first list argument.

Translating the definition of *reverse* into our language, and applying the *steps* function, gives the following step count equation:

$$\begin{array}{l} \mathit{steps}\ (\mathit{App}\ \mathit{reverse}\ (\mathit{Val}\ \mathit{xs})) = \mathit{fst}\ (\mathit{foldrVal}\ g\ (10, \mathit{VNil})\ \mathit{xs}) \\ \quad \textbf{where}\ g\ z\ (s, \mathit{zs}) = (s + 21 * (\mathit{lengthVal}\ \mathit{zs}) + 34, \\ \qquad\qquad \mathit{eval}\ (\mathit{App}\ (\mathit{App}\ \mathit{append}\ (\mathit{Val}\ \mathit{zs}))\ (\mathit{Val}\ (\mathit{VCons}\ z\ \mathit{VNil})))\ [\,]) \end{array}$$

This equation now gives a precise measure of the number of steps required to reverse a list. The final step in this example is to rewrite this equation into a closed form, purely in terms of the length of the argument list, without reference to the evaluation function *eval*. First of all, we can observe from the above definition that reversing the empty list incurs a cost of 10 steps. In turn, a non-empty list incurs a cost of 34 steps plus the current length of the reversed list multiplied by 21. Finally, the length of the reversed list increases by one for each recursive call. In conclusion, the number

of steps to reverse a list is given by a sum of the form:

$$10 + \sum_{x=0}^{length\ xs-1} 21x + 34$$

The correctness of the resulting cost expression with respect to the original step count function above can be verified by induction on the structure of the argument list. Expanding the resulting sum gives the following expression:

$$10 + 34 * (length\ xs) + \frac{21 * (length\ xs - 1) * (length\ xs)}{2}$$

That is, the number of steps is quadratic on the length of the input list. Once again, this is the result we expect, but we now see the constant factors involved.

Fast reverse

The reverse function can also be expressed using an accumulator:

$$\begin{array}{lcl} fastrev\ [\,]\ a & = & a \\ fastrev\ (x : xs)\ a & = & fastrev\ xs\ (x : a) \end{array} \quad \Leftrightarrow \quad \begin{array}{l} fastrev\ xs = foldr\ f\ id\ xs\ [\,] \\ \quad \textbf{where}\ f\ x\ g\ a = g\ (x : a) \end{array}$$

This definition should have better time performance because, as shown above, the time required to evaluate the *append* function is proportional to the length of the first argument, so appending the tail of the list is inefficient. The *steps* function produced for the accumulator version is directly proportional to the length of the list:

$$steps\ (App\ fastrev\ (Val\ xs)) = 15 + 26 * (lengthVal\ xs)$$

Therefore, the original quadratic time for reversing a list has been reduced to linear but, more importantly, we have made precise the constant factors involved. Re-expressing the accumulator version using fold-left, as given below, and applying the *steps* function, improves the time performance further:

$$reversel\ xs = foldl\ (\lambda a\ x \rightarrow x : a)\ [\,]\ xs$$
$$steps\ (App\ reversel\ (Val\ xs)) = 10 + 23 * (lengthVal\ xs)$$

Concatenation

Concatenating a list of lists can be defined by folding *append* over the list:

$$concat\ xs = foldr\ append\ [\,]\ xs$$

Using the step count for the *append* function, the resulting *steps* equation is:

$$\begin{aligned} &steps\ (App\ concat\ (Val\ xss)) = foldrVal\ f\ 10\ xss \\ &\quad \textbf{where}\ f\ ys\ s = 21 * (lengthVal\ ys) + 20 + s \end{aligned}$$

That is, the number of steps required in evaluation is the sum of the steps taken to apply the *append* function to each element in the list, plus the overhead for *concat* itself. If the argument to *concat* is a list where all the list elements are of the same length (which means that the number of steps taken in applying the *append* function will always be constant), then this can be simplified to:

$$\begin{aligned} steps\ (App\ concat\ (Val\ xss)) = 20 + \textbf{case}\ xss\ \textbf{of} \\ \qquad VNil \qquad\quad &\rightarrow 0 \\ \qquad VCons\ xs\ \_ &\rightarrow (lengthVal\ xss) * (21 * (lengthVal\ xs)) \end{aligned}$$

That is, the number of steps is now proportional to the length of the input list multiplied by the number of steps to evaluate appending an element of the list, which is proportional to the length of that element. Again, this is the form of result we expect, but the constant factors are now explicit.

The *concat* function can also be expressed using a fold-left, as follows:

$$concatl\ xss = foldl\ append\ [\,]\ xss$$

The complexity of the *append* function now depends on the length of its first argument, which is the accumulator, so this operation becomes more expensive as the *foldl* reaches the end of the argument list, resulting in quadratic time complexity:

$$\begin{aligned} &steps\ (App\ concat\ (Val\ xss)) = 10 + \textbf{case}\ xss\ \textbf{of} \\ &\quad VNil \qquad\quad \rightarrow 0 \\ &\quad VCons\ xs\ \_ \rightarrow 22 * (lengthVal\ xss) + \\ &\qquad 21 * (lengthVal\ xs) * (lengthVal\ xss - 1) * (lengthVal\ xss)\ /\ 2 \end{aligned}$$

Average

The arithmetic mean of a list of integers is defined as the sum of the list divided by the number of elements, which can be expressed as follows:

$$average\ xs = sum\ xs\ `div`\ length\ xs$$

This definition may be embedded in our language using a division operator with the expected semantics (and ignoring the issue of division by zero):

$$\begin{aligned} eval\ (Div\ x\ y)\ env = \mathbf{let}\ &(Const\ m) = eval\ x\ env \\ &(Const\ n) = eval\ y\ env \\ \mathbf{in}\ &Const\ (m\ `div`\ n) \end{aligned}$$

The corresponding steps function for the division case is:

$$steps'\ (Div\ x\ y)\ env = steps'\ x\ env + steps'\ y\ env + 3$$

Using these definitions, applying the *steps* function to the result of applying the *average* function to a list of integers gives the following cost equation:

$$steps\ (App\ average\ (Val\ xs)) = 27 + 42 * (length\ xs)$$

Note that although the *average* function gives a division by zero error when applied to the empty list, it takes 27 steps to reach this resolution.

The above definition for *average* requires two traversals of the list in order to calculate the result. However, the definition may be fused to only require a single traversal, by instead using fold-right to calculate a pair of the sum and length. This type of fusion is known as the "banana split property" of fold-right [53].

$$\begin{aligned} &averageFused\ xs = (\lambda a \rightarrow fst\ a\ `div`\ snd\ a) \\ &\quad (foldr\ (\lambda x\ a \rightarrow (x + fst\ a, 1 + snd\ a))\ (0, 0)\ xs) \end{aligned}$$

The cost of evaluating this definition (by first extending the language with support for pairs and the *fst* and *snd* functions to project from them), is the following function:

$$steps\ (App\ averageFused\ (Val\ xs)) = 23 + 33 * (length\ xs)$$

That is, the fused version of *average* is more efficient (in particular the number of steps

required for each element is 33 rather than 42), despite the overhead of projecting out and repairing the accumulator at each element.

4.4 Benchmarking

We conclude this chapter by considering how the performance results that are produced from our theory correspond to those from existing implementations. In order to achieve this, we benchmark our results against three well known implementations of Haskell: Hugs (WinHugs version May 2006), GHC and GHCi (both version 6.8.2). Although Haskell is a lazy language, all of our examples are used in a strict manner with fully evaluated arguments, and do not involve sharing. For reference, our benchmarks were produced on a 2.16GHz laptop with 2GB RAM.

4.4.1 Reduction counts

Hugs is an interpreter that can produce statistics on the number of reductions performed, number of cells allocated, and the number of garbage collections. The number of reductions gives an approximate measure of how much work the interpreter has done, but will also include the cost of printing the result to the terminal. To disregard this cost, we subtract the number of reductions that are required to print the result of an expression from the total number of reductions to evaluate the expression. For example, if summing a list $[1, 2, 3]$ takes 10 reductions, and printing its result (the value 6) takes 2 reductions, then we take the cost of the actual sum as 8 reductions.

We construct functions to express the number of reductions in Hugs by taking experimental results using the interpreter, matching to the basic structure of the expected function, and then extrapolating the results to a larger input and verifying that this value corresponds with the result from Hugs.

For each of the example functions considered in this chapter, the table below compares the result of applying our theory with the number of reductions in Hugs. For each function listed in the first column, the second column gives the number of

steps required according to our theory. In turn, the third column gives the number of reductions required using our original evaluation function. And finally, the fourth column gives the number of reductions required if the example function is implemented directly in Haskell. We can use these results to show the overhead of having an embedded language, rather than using functions expressed in Haskell directly.

Function	Theory (*steps*)	Reductions (*eval*)	Reductions (Haskell)
sum	$10 + 21n$	$12_{1.2} + 39_{1.9}n$	$4_{0.4} + 3_{0.14}n$
sumAcc	$15 + 26n$	$22_{1.5} + 56_{2.2}n$	$3_{0.2} + 6_{0.23}n$
append	$15 + 21n$	$28_{1.3} + 31_{1.5}n$	$2_{0.13} + 1_{0.05}n$
concat	$20 + 21nm$	$14_{0.7} + 32n + 31_{1.5}nm$	$2_{0.1} + 3n + 1_{0.05}nm$
reverse	$10 + 34n + \frac{21n(n-1)}{2}$	$14_{1.4} + 58_{1.7}n + \frac{31_{1.5}n(n-1)}{2}$	$2_{0.2} + 3_{0.09}n + \frac{1_{0.05}n(n-1)}{2}$
fastrev	$15 + 26n$	$24_{1.6} + 48_{1.8}n$	$3_{0.2} + 2_{0.07}n$
average	$27 + 42n$	$48_{1.8} + 67_{1.6}n$	$61_{2.26} + 12_{0.29}n$
averageFused	$23 + 33n$	$41_{1.8} + 60_{1.8}n$	$58_{2.52} + 18_{0.55}n$

In this table, n is the length of the argument list, and in the case of *concat*, m is the length of the component lists. The numeric subscripts are not part of the formulae themselves, but are included to provide an indication of the fit between our theory and the number of reductions in Hugs. For example, if we consider comparing the formulae $10 + 21n$ and $12_{1.2} + 39_{1.9}n$ in the first row of the table, the subscript 1.2 indicates that there are 1.2 initial reductions for each initial step in our theory, which is calculated simply by dividing 12 by 10. Similarly, the subscript 1.9 indicates that there are 1.9 further reductions for each further step in our theory, given by dividing 39 by 21. If these subscript values are the same across different examples, then there is a direct correspondence between steps in our theory and reductions in Hugs.

The results show that the step counts in our theory are mostly more coarse-grained than the reductions in Hugs, with one step typically corresponding to an average of 1.6 reductions. In contrast, the step counts are typically more fine-grained for the functions expressed directly in Haskell, with one step averaging 0.44 reductions. There are some instances where the number of reductions for the Haskell definitions

are considerably lower (*append*, *concat*, *reverse*, *fastreverse*), which are most likely explained by optimisations occurring in Hugs' implementation of lists.

4.4.2 Execution time

GHC can be used either as an interactive interpreter (GHCi), similar to Hugs, or as a compiler. Both of these implementations can output execution times: GHCi can print the execution time required to evaluate an expression to 20ms accuracy, and GHC can compile the input program to additionally run the profiler described in section 1.6.2, which outputs execution time information. To only consider the execution costs of applying the *eval* function, and ignore other overheads, we can enclose a "cost-centre" around this function when we use the profiler.

As we are now measuring the execution time rather than the number of reductions, we need the evaluation to take a significant amount of time. Hence we either need to call the example functions with a large input size, or repeat the evaluation a number of times and divide the resulting execution time by that amount. Calling our examples with a large enough input to give a reasonably sized execution time can exceed the maximum stack size for GHC, so we instead adopt the latter approach.

For each of the example functions the table below compares the result of applying our theory with the execution time in seconds using GHC, where the execution times are produced by timing each of the example functions ten million times. For each function listed in the first column, the second column gives the number of steps required according to our theory. In turn, the third column gives the execution time using GHCi, and finally the fourth column gives the execution time in GHC itself.

Function	Theory (*steps*)	GHCi	GHC
sum	$21n$	$156_{7.4}n$	$8.1_{0.39}n$
sumAcc	$26n$	$198_{7.6}n$	$11.7_{0.45}n$
append	$21n$	$178_{8.4}n + 21m$	$7.5_{0.38}n + 0.3m$
concat	$21nm$	$14.5n + 160_{7.6}nm$	$0.72n + 8.8_{0.42}nm$
reverse	$34n + \frac{21n(n-1)}{2}$	$272_{7.9}n + \frac{133_{6.3}n(n-1)}{2}$	$145n + \frac{6.8n(n-1)}{2}$
fastrev	$26n$	$208_{8}n$	$12.6_{0.48}n$
average	$42n$	$284_{6.8}n$	$14.9_{0.35}n$
averageFused	$33n$	$214_{6.5}n$	$11.6_{0.35}n$

In this table, n is the length of the argument list, while m is the length of the second list in the case of *append*, and the length of the component lists in the case of *concat*. It proved difficult to determine the standalone constants in the resulting formulae (e.g. the constant a in $a + bn$), so we only consider the non-constant terms in the table. Once again, the numeric subscripted values are not part of the formulae themselves, but are included to illustrate the fit between our theory and the execution times in GHC. For example, if we consider comparing the formula $21n$ and $156_{7.4}n$ in the first row of the table, the subscript 7.4 indicates that there are 7.4 ($\times 10^{-7}$) seconds for each step in our theory, given by dividing 156 ($\times 10^{-7}$) by 21.

The results show that each step in our theory corresponds to an average of 7×10^{-7} seconds in GHCi, and 0.4×10^{-7} seconds in GHC. While in the case of reduction counts in the previous section where there were some considerable deviations from the average, the execution times in GHC are more closely aligned with our theory. Although it is difficult to give definitive reasons for this correlation, one possible explanation is that reductions in Hugs may involve an arbitrary amount of work, whereas execution times give a precise measure of the amount of work required. Moreover, GHC has a sophisticated strictness analyzer, which can transform strict functions into strict assembly language without any of the overhead of lazy evaluation [54].

4.5 Summary

In this chapter we have presented a technique for instrumenting an abstract machine with time information based on the number of transitions that are performed, and showed how the resulting machine can then be transformed into a high-level function that measures time performance. All the steps in the derivation process are purely calculational, in that the required function at each stage is calculated directly from a specification of its desired behaviour.

To illustrate the use of this technique in practice we considered a number of example functions that operate on lists, and showed how to derive formulae that capture their time complexity in terms of an underlying abstract machine. Being able to derive precise formulae, rather than asymptotic bounds, allows us to observe the differences between functions in the same complexity class, and observe the overheads of techniques such as accumulation or tupling. The section concluded by benchmarking our performance results using Hugs and GHC, in order to show how well our approach to time analysis corresponds to implementations of Haskell. The next chapter will follow a similar programme, but instead focus on space rather than time.

Chapter 5

Adding space

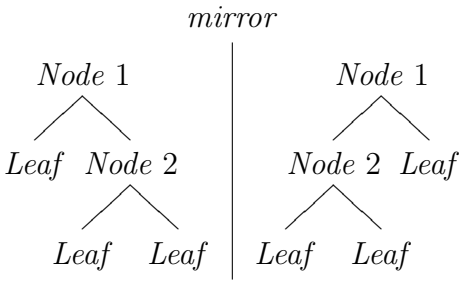
This chapter explores how an abstract machine can be instrumented with space requirements, in such a way that it can be transformed into a high-level function that measures space usage from the abstract machine level. This function will then be used to examine the space behaviour of some example programs.

5.1 Mirroring a tree

By way of a motivating example for the use of abstract machines in the study of space behaviour, let us consider the problem of mirroring a tree, as discussed in [55]. We first define a datatype of binary trees with integers stored at the nodes:

data *Tree* = *Leaf* | *Node Tree Int Tree*

A tree is mirrored by swapping the branches of every node. As an example, applying the mirror function to the tree *Node Leaf* 1 (*Node Leaf* 2 *Leaf*), gives the resulting tree *Node* (*Node Leaf* 2 *Leaf*) 1 *Leaf*, which is more apparent pictorially:



Mirroring a tree can be defined directly over the structure of the tree:

$$\begin{aligned} &\mathit{mirror} && :: \mathit{Tree} \to \mathit{Tree} \\ &\mathit{mirror}\ \mathit{Leaf} && = \mathit{Leaf} \\ &\mathit{mirror}\ (\mathit{Node}\ l\ x\ r) && = \mathit{Node}\ (\mathit{mirror}\ r)\ x\ (\mathit{mirror}\ l) \end{aligned}$$

That is, a *Leaf* is already mirrored, and a *Node* is mirrored by swapping its mirrored branches. In this definition, the semantics of the mirror function is immediately apparent, since it is defined using simple recursion. The definition is also concise, and does not require any additional data structures at the program level. Because of this, however, the space requirements are obscured: we don't know in what order the recursive calls are occurring in, or how to measure the space requirements of the partial applications occurring in the expression $(\mathit{Node}\ (\mathit{mirror}'\ r)\ x)\ (\mathit{mirror}'\ l)$.

An alternative definition for the *mirror* function utilizes a *zipper* [56], which provides an efficient way for navigating around tree-like structures. The zipper data structure for the tree type given above consists of a constructor *Top* to mark the top of the tree, a constructor *NodeL* to represent traversing the left branch, storing the integer value and right branch, and another constructor *NodeR* to traverse the right branch, storing the left branch and integer value:

$$\textbf{data}\ \mathit{Zipper} = \mathit{Top} \mid \mathit{NodeL}\ \mathit{Zipper}\ \mathit{Int}\ \mathit{Tree} \mid \mathit{NodeR}\ \mathit{Tree}\ \mathit{Int}\ \mathit{Zipper}$$

The *mirror* function can be defined using the zipper by first navigating to the leftmost subtree, and mirroring from the bottom of the tree upwards. We define the function to mirror a tree by initially loading with the empty zipper *Top*:

$$mirror'\ t = load\ t\ Top$$

The *load* function locates the leftmost *Leaf* in a tree, whilst storing the integer values and the right trees encountered, and then executes the *unload* function:

$$\begin{aligned}
&load && :: Tree \to Zipper \to Tree \\
&load\ (Node\ l\ n\ r)\ c &&= load\ l\ (NodeL\ c\ n\ r) \\
&load\ Leaf\ c &&= unload\ Leaf\ c
\end{aligned}$$

In the *Nodel* case, the *unload* function calls *load* on the right subtree, in order to mirror it. In turn, in the *NodeR* case the *unload* function swaps the left and right subtrees in order to mirror the node:

$$\begin{aligned}
&unload && :: Tree \to Zipper \to Tree \\
&unload\ t\ Top &&= t \\
&unload\ l\ (NodeL\ c\ n\ r) &&= load\ r\ (NodeR\ l\ n\ c) \\
&unload\ r\ (NodeR\ l\ n\ c) &&= unload\ (Node\ r\ n\ l)\ c
\end{aligned}$$

The evaluation trace for calling *mirror′* on the example tree given earlier is:

$$\begin{aligned}
&= mirror'\ (Node\ Leaf\ 1\ (Node\ Leaf\ 2\ Leaf)) \\
&= load\ (Node\ Leaf\ 1\ (Node\ Leaf\ 2\ Leaf))\ Top \\
&= load\ Leaf\ (NodeL\ Top\ 1\ (Node\ Leaf\ 2\ Leaf)) \\
&= unload\ Leaf\ (NodeL\ Top\ 1\ (Node\ Leaf\ 2\ Leaf)) \\
&= load\ (Node\ Leaf\ 2\ Leaf)\ (NodeR\ Leaf\ 1\ Top) \\
&= load\ Leaf\ (NodeL\ (NodeR\ Leaf\ 1\ Top)\ 2\ Leaf) \\
&= unload\ Leaf\ (NodeL\ (NodeR\ Leaf\ 1\ Top)\ 2\ Leaf) \\
&= load\ Leaf\ (NodeR\ Leaf\ 2\ (NodeR\ Leaf\ 1\ Top)) \\
&= unload\ Leaf\ (NodeR\ Leaf\ 2\ (NodeR\ Leaf\ 1\ Top)) \\
&= unload\ (Node\ Leaf\ 2\ Leaf)\ (NodeR\ Leaf\ 1\ Top) \\
&= unload\ (Node\ (Node\ Leaf\ 2\ Leaf)\ 1\ Leaf)\ Top \\
&= Node\ (Node\ Leaf\ 2\ Leaf)\ 1\ Leaf
\end{aligned}$$

How much space does this definition for the mirror function require? Looking at the shape of the evaluation trace indicates that *mirror′* executes in constant space, but we

can also estimate the space requirements by looking at the structure of the function definitions. By comparing the left and right sides of each function, and assuming we can re-use space for constructors, there is no line of the program where the right hand side is larger than the left: only the names of the constructors differ on either side of the functions. If we can perform this in-place update then the *mirror* function should not require any additional space for evaluation.

We now have two definition for the mirror function. The original function, *mirror*, has the advantage of being clear and concise, and directly implements the high-level specification for mirroring a tree. In contrast, the function $mirror'$ reveals the low level execution details, and shows that mirroring can potentially be achieved in constant space. Hence, both definitions of mirror have their advantages and it would be useful to combine these two different views, in order that a programmer could write the high-level version, but may expect it to execute as in the low-level version, in particular for the purposes of considering space usage.

If we take the original definition for the *mirror* function and transform it into continuation-passing style and defunctionalize the resulting continuation, the result is precisely the zipper version, which is an abstract machine. In this way, we can combine the advantages of the clarity of a high-level definition with being able to access and measure the low-level execution details.

5.2 Measuring space

Chapter 3 showed how a low-level abstract machine can be derived from a high-level evaluator for a language, and now we consider how the resulting machine can be used to study space usage. In order to use an abstract machine to measure the space requirements of evaluation, we require a way of instrumenting the machine with space information. This information will be based on the sizes of the data structures built by the machine during its execution. In the case of the machine derived in section 3.3, the data structures are pairs consisting of either an expression and a control stack in the *eval* function, or a control stack and a value in the *exec* function. To recap, the

complete machine is defined by

$$
\begin{aligned}
&\mathit{evaluate} && :: \mathit{Expr} \to \mathit{Value}\\
&\mathit{evaluate}\ e && = \mathit{evalMach}\ (e, [\,])\\
&\mathit{evalMach} && :: (\mathit{Expr}, \mathit{Stack}) \to \mathit{Value}\\
&\mathit{evalMach}\ (\mathit{Val}\ v, \quad s) && = \mathit{exec}\ (s, v)\\
&\mathit{evalMach}\ (\mathit{Add}\ x\ y, s) && = \mathit{evalMach}\ (x, \mathit{EVAL}\ y : s)\\
&\mathit{exec} && :: (\mathit{Stack}, \mathit{Value}) \to \mathit{Value}\\
&\mathit{exec}\ ([\,], \qquad v) && = v\\
&\mathit{exec}\ (\mathit{EVAL}\ y : s, n) && = \mathit{evalMach}\ (y, \mathit{ADD}\ n : s)\\
&\mathit{exec}\ (\mathit{ADD}\ n : s,\ n') && = \mathit{exec}\ (s, n + n')
\end{aligned}
$$

where the relevant types are defined as follows:

$$
\begin{aligned}
&\textbf{data}\ \mathit{Expr} && = \mathit{Val}\ \mathit{Int} \mid \mathit{Add}\ \mathit{Expr}\ \mathit{Expr}\\
&\textbf{type}\ \mathit{Value} && = \mathit{Int}\\
&\textbf{type}\ \mathit{Stack} && = [\mathit{Op}]\\
&\textbf{data}\ \mathit{Op} && = \mathit{EVAL}\ \mathit{Expr} \mid \mathit{ADD}\ \mathit{Value}
\end{aligned}
$$

Computing sizes

In order to measure the sizes of data structures, we define a size function on each data structure required in the machine, where the size of structures is measured simply by counting constructors. We introduce this as a type class, *Sized*, which has a single method *size* that takes a data structure and returns its size as an integer:

$$
\begin{aligned}
&\textbf{class}\ \mathit{Sized}\ a\ \textbf{where}\\
&\quad \mathit{size} :: a \to \mathit{Int}
\end{aligned}
$$

We then define this size function on pairs, lists of sized structures, expressions, operations and integers, by simply counting the number of constructors in each structure, with an integer value assumed to have unit size:

$$
\begin{aligned}
&\textbf{instance}\ (\mathit{Sized}\ a, \mathit{Sized}\ b) \Rightarrow \mathit{Sized}\ (a, b)\ \textbf{where}\\
&\quad \mathit{size}\ (a, b) && = 1 + \mathit{size}\ a + \mathit{size}\ b
\end{aligned}
$$

```
instance Sized a ⇒ Sized [a] where
  size []       = 1
  size (x : xs) = 1 + size x + size xs
instance Sized Expr where
  size (Val v)   = 1 + size v
  size (Add x y) = 1 + size x + size y
instance Sized Op where
  size (EVAL e) = 1 + size e
  size (ADD v)  = 1 + size v
instance Sized Int where
  size n        = 1
```

Space semantics

In a similar manner to [25], we define the space requirements to evaluate an expression to be the maximum-sized data structure generated during the running of the machine. This approach assumes that there is a garbage collector which identifies any space that has been freed at each transition, and allows it to be re-used at a later point.

Consider a reduction sequence that starts with the configuration $x_0 = (e, [])$ and ends with the value v:

$$x_0 \to x_1 \to \cdots \to x_n \to v$$

The maximum-sized data structure produced during the reduction sequence (ignoring the fact that elements of this list may have different types), can be calculated by mapping *size* over every configuration and then applying *maximum*:

$$(maximum \circ map\ size)\ [x_0, x_1, \cdots x_n, v]$$

To calculate the space requirements, however, we should not include the space occupied by the expression e, but instead only measure the additional space required for evaluation. The justification for this approach is that the space the expression is

occupying has already been allocated before the machine is started. In conclusion, the space requirements can be captured as follows:

$$((maximum \circ map\ size)\ [x_0, x_1, \cdots x_n, v]) - size\ e$$

Initial approach

Now we consider how to implement our space semantics in an abstract machine. The semantics can be described separately from the details of the machine by only modeling the data that the machine is aware of at any transition (i.e. the configurations on either side of the transition) in order to concentrate on how the instrumentation is performed. We first define a function that takes a list whose elements represent machine configurations, and pairs them into successive transitions:

$$\begin{aligned} &pairs && :: [a] \to [(a, a)] \\ &pairs\ [x] && = [\,] \\ &pairs\ (x : y : ys) && = (x, y) : pairs\ (y : ys) \end{aligned}$$

We instrument the machine similarly to the previous chapter by using an accumulator that is updated at every transition, which can be modeled as a fold-left over the result of applying *pairs* to a list of configurations. Using this idea, the space semantics from the previous section may be expressed by taking the size of the first configuration as the initial accumulator, and updating the accumulator with any right-hand side configuration that is greater in size (where $\uparrow$ is the maximum operator):

$$maximumSize\ xs = foldl\ (\lambda m\ (x, y) \to m \uparrow (size\ y))\ (size\ (head\ xs))\ (pairs\ xs)$$

The overall space requirements can thus be obtained by applying the *maximumSize* function and then subtracting the size of the original expression. The abstract machine given at the start of the chapter can be instrumented directly with this semantics as shown below, by introducing an accumulator that is initialised with the size of the starting configuration, and updated in the appropriate manner on each transition. While the instrumented machine does not explicitly use *maximumSize*, it will be useful later on when we consider refining the instrumentation.

$$
\begin{array}{lll}
\mathit{space} & :: & \mathit{Expr} \to \mathit{Int} \\
\mathit{space}\ e & = & \textbf{let}\ x = (e, [\,])\ \textbf{in}\ \mathit{spaceMach}\ x\ (\mathit{size}\ x) - \mathit{size}\ x \\
\mathit{spaceMach} & :: & (\mathit{Expr}, \mathit{Stack}) \to \mathit{Int} \to \mathit{Int} \\
\mathit{spaceMach}\ (\mathit{Val}\ v, s)\quad m & = & \textbf{let}\ x = (s, v) \\
 & & \textbf{in}\ \mathit{exec}'\ x\ (m \uparrow \mathit{size}\ x) \\
\mathit{spaceMach}\ (\mathit{Add}\ x'\ y, s)\ m & = & \textbf{let}\ x = (x', \mathit{EVAL}\ y : s) \\
 & & \textbf{in}\ \mathit{spaceMach}\ x\ (m \uparrow \mathit{size}\ x) \\
\mathit{exec}' & :: & (\mathit{Stack}, \mathit{Value}) \to \mathit{Int} \to \mathit{Int} \\
\mathit{exec}'\ ([\,], \quad v)\quad m & = & m \uparrow \mathit{size}\ v \\
\mathit{exec}'\ (\mathit{EVAL}\ y : s, n)\quad m & = & \textbf{let}\ x = (y, \mathit{ADD}\ n : s) \\
 & & \textbf{in}\ \mathit{spaceMach}\ x\ (m \uparrow \mathit{size}\ x) \\
\mathit{exec}'\ (\mathit{ADD}\ n : s,\ n')\quad m & = & \textbf{let}\ x = (s, n + n') \\
 & & \textbf{in}\ \mathit{exec}'\ x\ (m \uparrow \mathit{size}\ x)
\end{array}
$$

There is, however, a problem with this instrumented machine. In particular, it results in expressions that measure the size of the control stack. For example, the case for *Val* v in the *spaceMach* function requires the size of the control stack s. The next stage in producing an evaluator from this instrumented machine is to refunctionalize the stack, turning it back into a function. Any expressions that measure the size of the original stack data structure will be replaced with expressions that measure the size of a function, which is not readily supported in our current framework of counting constructors. To avoid this problem, we now aim to re-express the instrumentation so that it does not directly measure the size of the stack.

Transition approach

In the abstract machine the stack is threaded throughout both the *eval* and *exec* functions, and therefore always appears on both sides of a transition, so any explicit references could be eliminated by instead only measuring how the stack changes. Annotating each transition with the difference in the size of the right and left config-

urations would remove the need to measure the size of the stack directly:

$$x_0 \xrightarrow{s_0} x_1 \xrightarrow{s_1} \cdots \xrightarrow{s_n-1} x_n \xrightarrow{s_n} v$$
$$\textbf{where } s_n = \mathit{size}\ x_{n+1} - \mathit{size}\ x_n$$

The difference in size s_n across a transition $x_n \rightarrow x_{n+1}$ is calculated by subtracting the size of x_n from that of x_{n+1}. As the machine runs, the space occupied by the current configuration grows and shrinks according to the transition rule being applied. Using the *size* function, each transition can be annotated as either increasing or decreasing the space, according to the difference in size between the data structures on the right- and left-hand sides of the transition, as shown below:

$$\begin{array}{lll}
\mathit{evaluate} & :: & \mathit{Expr} \rightarrow \mathit{Value} \\
\mathit{evaluate}\ e & \overset{2}{=} & \mathit{eval}\ (e, [\,]) \\
\mathit{eval} & :: & (\mathit{Expr}, \mathit{Stack}) \rightarrow \mathit{Value} \\
\mathit{eval}\ (\mathit{Val}\ v, \quad s) & \overset{-1}{=} & \mathit{exec}\ (s, v) \\
\mathit{eval}\ (\mathit{Add}\ x\ y, \quad s) & \overset{1}{=} & \mathit{eval}\ (x, \mathit{EVAL}\ y : s) \\
\mathit{exec} & :: & (\mathit{Stack}, \mathit{Value}) \rightarrow \mathit{Value} \\
\mathit{exec}\ ([\,], \quad v) & \overset{-2}{=} & v \\
\mathit{exec}\ (\mathit{EVAL}\ y : s, m) & \overset{0}{=} & \mathit{eval}\ (y, \mathit{ADD}\ m : s) \\
\mathit{exec}\ (\mathit{ADD}\ m : s, n) & \overset{-3}{=} & \mathit{exec}\ (s, m + n)
\end{array}$$

For example, the transition for evaluating an addition (i.e. the second equation for *eval*) is labeled 1 because $\mathit{size}\ (x, \mathit{EVAL}\ y : s) - \mathit{size}\ (\mathit{Add}\ x\ y, s) = 1$. This is the only transition where the space usage increases, with the exception of the initial *evaluate* transition. Note that if we had represented the stack type directly as a recursive datatype, rather than indirectly using the Haskell list type (as discussed in section 3.3), then there would be no size-increasing transitions. For the purposes of this section, however, we prefer the above definition that includes both positive and negative transition sizes. When we consider extending the language we will return to a recursive stack datatype, in order to be more space efficient.

To exploit these transition differences, our space semantics needs to be modified

to avoid directly measuring the size of the right configuration. We first introduce a function *lastSize* that returns the size of the last element in a list. Similarly to the function *maximumSize* defined earlier, we define *lastSize* using a fold-left over pairs of configurations, taking the size of the first configuration as the initial accumulator, and successively updating this value with the size of each right-hand configuration:

$$lastSize\ xs = foldl\ (\lambda l\ (x, y) \rightarrow size\ y)\ (size\ (head\ xs))\ (pairs\ xs)$$

By construction, the second component of a pair produced by the function *pairs* is always equal to the first component of the next pair, which in the above definition for *lastSize* means that the value l will always be equal to $size\ x$. Hence, we can rewrite *lastSize* to instead calculate $size\ y$ by adding l and $(size\ y) - (size\ x)$, as follows:

$$lastSize'\ xs = foldl\ (\lambda l\ (x, y) \rightarrow l + (size\ y - size\ x))\ (size\ (head\ xs))\ (pairs\ xs)$$

We can then calculate a new function *lastMaxSize* that returns a pair of the size of the last element and the maximum, using the specification

$$lastMaxSize\ xs = (lastSize'\ xs, maximumSize\ xs)$$

and then apply fusion to give the following definition:

$$\begin{aligned} &lastMaxSize\ xs = foldl\ f\ (size\ (head\ xs), size\ (head\ xs))\ (pairs\ xs) \\ &\quad \textbf{where}\ f\ (l, m)\ (x, y) = (l + (size\ y - size\ x), m \uparrow size\ y) \end{aligned}$$

The final stage is to substitute $size\ y$ for the equivalent $l + (size\ y - size\ x)$, to give a semantics that only uses the difference in transition sizes. Recall that the overall space requirements are the maximum expression size minus the size of the starting expression. The required subtraction can be incorporated by initialising the accumulator to the size of the starting configuration minus the size of the expression component.

The derivation of this semantics has shown that our accumulator requires two components in order to capture space requirements. We refine this semantics by now taking a more intuitive memory management approach, but refer back to the derived version as our definition for the notion of space requirements.

5.3 Memory management

In this section we show how to instrument the machine with space requirements by first introducing a data structure that models a memory manager that keeps track of the space usage, and consists of a pair of non-negative integers:

$$\textbf{type}\ \mathit{MemMng} = (\mathit{Int}, \mathit{Int})$$

The first component is the amount of memory that has been explicitly freed at the current point, and the second is the amount that has been explicitly allocated:

$$\begin{aligned}
&\mathit{freed} &&:: \mathit{MemMng} \to \mathit{Int}\\
&\mathit{freed} &&= \mathit{fst}\\
&\mathit{allocated} &&:: \mathit{MemMng} \to \mathit{Int}\\
&\mathit{allocated} &&= \mathit{snd}
\end{aligned}$$

Both components are necessary to capture an accurate space model, in that memory freed by earlier evaluation may be re-used later on. The rationale for this approach is that whether a transition has a positive or negative size difference has more than just opposite effects on the space usage. In particular, a negative size difference only frees up memory, whereas a positive difference requires more memory, which can either be satisfied from the free memory available, or a request for additional memory.

Two functions are defined on the manager to allocate and free memory, *alloc* and *free*. To free some memory, the amount to be freed is simply added to the free memory integer, and is then available to use in later allocation requests:

$$\begin{aligned}
&\mathit{free} &&:: \mathit{Int} \to \mathit{MemMng} \to \mathit{MemMng}\\
&\mathit{free}\ n\ (f, a) &&= (f + n, a)
\end{aligned}$$

When allocating memory, the request is first satisfied using the pool of free memory that is currently available, by subtracting the amount from the free memory integer until it is zero, with the difference then added to the allocated memory:

$$\begin{aligned}
&\mathit{alloc} &&:: \mathit{Int} \to \mathit{MemMng} \to \mathit{MemMng}\\
&\mathit{alloc}\ n\ (f, a) &&= (f \dot{-} n, a + (n \dot{-} f))
\end{aligned}$$

For simplicity we assume an infinite amount of memory, and that allocation requests are always successful. The auxiliary subtraction function, $x \dot{-} y$, is defined as the maximum of $x - y$ and 0, thereby ensuring that the result is never negative:

$$
\begin{aligned}
&(\dot{-}) \quad :: \mathit{Int} \to \mathit{Int} \to \mathit{Int} \\
&x \dot{-} y = (x - y) \uparrow 0
\end{aligned}
$$

We define an initial memory manager as having no free or allocated amounts:

$$
\begin{aligned}
&\mathit{initial} :: \mathit{MemMng} \\
&\mathit{initial} = (0, 0)
\end{aligned}
$$

For the purposes of later proofs, we will exploit the following properties for these functions, which can easily be proved from the above definitions:

$$
\begin{aligned}
\mathit{free}\ n \circ \mathit{free}\ m &= \mathit{free}\ (m + n) && (1) \\
\mathit{alloc}\ n \circ \mathit{alloc}\ m &= \mathit{alloc}\ (m + n) && (2) \\
\mathit{alloc}\ n \circ \mathit{free}\ n &= \mathit{id} && (3) \\
\mathit{allocated} \circ \mathit{free}\ n &= \mathit{allocated} && (4)
\end{aligned}
$$

The first and second properties express that repeated occurrences of *free* or *alloc* may be accumulated. The third states that a *free* immediately followed by an *alloc* of the same amount has no effect, because the allocation can use up the previously freed amount. Finally, the last property expresses that freeing memory does not affect the amount allocated.

Transition costs

To add space information to the abstract machine we require a way of instrumenting each transition with its cost. The space requirements are added using an accumulator, so that it remains an abstract machine. The accumulator itself takes the form of a memory manager, and is updated according to the sizes on either side of the transition. For a transition of the form $x \to y$, the cost of the transition, written $\mathit{cost}\ x_s\ y_s$, depends on the size of the data structure on the left-hand side, x_s, and right-hand

side, y_s. The *cost* function captures the idea that as much space as possible is re-used, and is implemented by first freeing the space occupied by structures in x that do not occur in y, allowing it to be re-used, and then allocating the space required for additional structures that only occur in y:

$$\begin{aligned} &\mathit{cost} \quad :: \mathit{Int} \to \mathit{Int} \to \mathit{MemMng} \to \mathit{MemMng} \\ &\mathit{cost}\ x_s\ y_s = \mathit{alloc}\ (y_s \dot{-} x_s) \circ \mathit{free}\ (x_s \dot{-} y_s) \end{aligned}$$

Calling the *cost* function with two sizes causes one side of the composition to collapse, leaving either a *free* or an *alloc*. For example, *cost* 1 3 gives the result *alloc* 2, whereas *cost* 3 1 gives the result *free* 2.

5.3.1 Correctness of semantics

We can prove that our new semantics is correct with respect to the derived space semantics in the previous section by first expressing it as a fold-left over a list of machine configurations. For the starting accumulator, we take the *initial* memory manager with any initial data structures allocated (in this case the size of the empty list), and for the operator we take the *cost* function:

$$\begin{aligned} &\mathit{costs}\ xs = \mathit{foldl}\ (\lambda m\ (x, y) \to \mathit{cost}\ (\mathit{size}\ x)\ (\mathit{size}\ y)\ m)\ m'\ (\mathit{pairs}\ xs) \\ &\quad \textbf{where}\ m' = \mathit{alloc}\ (\mathit{size}\ [\,])\ \mathit{initial} \end{aligned}$$

The second component of the result returned by *lastMaxSize* (with the revised starting accumulator) is equal to the allocated amount returned by *costs*, whilst the first component, which represents the size of the current configuration, is equal to the allocated amount minus the freed amount returned by *costs*. This property may be proved by induction over the structure of the non-empty list xs, and then by case analysis for when x is less than y, and vice versa.

5.4 Space function

The abstract machine can now be extended with the memory manager, by threading the required pair through the machine and updating it on the right hand side.

With the definition of *cost* expanded and simplified and using the memory manager properties given earlier, the resulting machine is as follows:

$$
\begin{aligned}
&\mathit{spaceMach} && :: (\mathit{Expr}, \mathit{Stack}, \mathit{MemMng}) \to (\mathit{Value}, \mathit{MemMng}) \\
&\mathit{spaceMach}\ (\mathit{Val}\ v, s, m) && = \mathit{exec}'\ (s, v, \mathit{free}\ 1\ m) \\
&\mathit{spaceMach}\ (\mathit{Add}\ x\ y, s, m) && = \mathit{spaceMach}\ (x, \mathit{EVAL}\ y : s, \mathit{alloc}\ 1\ m) \\
&\mathit{exec}' && :: (\mathit{Stack}, \mathit{Value}, \mathit{MemMng}) \to (\mathit{Value}, \mathit{MemMng}) \\
&\mathit{exec}'\ ([\,], v, m) && = (v, \mathit{free}\ 2\ m) \\
&\mathit{exec}'\ (\mathit{EVAL}\ y : s, n, m) && = \mathit{spaceMach}\ (y, \mathit{ADD}\ n : s, m) \\
&\mathit{exec}'\ (\mathit{ADD}\ n : s, n', m) && = \mathit{exec}'\ (s, n + n', \mathit{free}\ 3\ m)
\end{aligned}
$$

The use of the memory management functions *free* and *alloc* in this new machine corresponds directly to the size labels on the transition function given in section 5.2. For example, in the *Val* case above the use of $\mathit{free}\ 1\ m$ implements the size label -1 in the corresponding case for the labeled transition function. Similarly, the starting configuration first allocates two units of space from an initial memory manager to hold the empty stack pair constructor, corresponding to the size label 2 in the labeled transition function:

$$\mathit{space}\ e = \mathit{spaceMach}\ (e, [\,], \mathit{alloc}\ 2\ \mathit{initial})$$

Refunctionalize

We now wish to derive an evaluator based upon this new machine. The first stage in this process is to refunctionalize the control stack. The new evaluator can be calculated starting from the following specification:

$$\mathit{spaceTail}\ (e, m)\ (\mathit{exec}'\ s) = \mathit{spaceMach}\ (e, s, m)$$

This equation states that *spaceTail* gives the same result as *spaceMach*, except it takes a continuation, instead of a context. The new definition can be calculated by induction over the expression e. Note that, for simplicity, the exec' function is written in curried form in both the specification above and the calculation below.

Case : $e = \mathit{Val}\ v$

$$
\begin{array}{ll}
 & \mathit{spaceTail}\ (\mathit{Val}\ v, m)\ (\mathit{exec}'\ s) \\
= & \quad \{\ \text{specification}\ \} \\
 & \mathit{spaceMach}\ (\mathit{Val}\ v, s, m) \\
= & \quad \{\ \text{definition of } \mathit{spaceMach}\ \} \\
 & \mathit{exec}'\ s\ (v, \mathit{free}\ 1\ m)
\end{array}
$$

Case : $e = \mathit{Add}\ x\ y$

$$
\begin{array}{ll}
 & \mathit{spaceTail}\ (\mathit{Add}\ x\ y, m)\ (\mathit{exec}'\ s) \\
= & \quad \{\ \text{specification}\ \} \\
 & \mathit{spaceMach}\ (\mathit{Add}\ x\ y, s, m) \\
= & \quad \{\ \text{definition of } \mathit{spaceMach}\ \} \\
 & \mathit{spaceMach}\ (x, \mathit{EVAL}\ y : s, \mathit{alloc}\ 1\ m) \\
= & \quad \{\ \text{inductive assumption}\ \} \\
 & \mathit{spaceTail}\ (x, \mathit{alloc}\ 1\ m)\ (\mathit{exec}'\ (\mathit{EVAL}\ y : s)) \\
= & \quad \{\ \text{definition of } \mathit{exec}'\ \} \\
 & \mathit{spaceTail}\ (x, \mathit{alloc}\ 1\ m)\ (\lambda(n, m') \rightarrow \mathit{spaceMach}\ (y, \mathit{ADD}\ n : s, m')) \\
= & \quad \{\ \text{inductive assumption}\ \} \\
 & \mathit{spaceTail}\ (x, \mathit{alloc}\ 1\ m)\ (\lambda(n, m') \rightarrow \mathit{spaceTail}\ (y, m')\ (\mathit{exec}'\ (\mathit{ADD}\ n : s))) \\
= & \quad \{\ \text{definition of } \mathit{exec}'\ \} \\
 & \mathit{spaceTail}\ (x, \mathit{alloc}\ 1\ m)\ (\lambda(n, m') \rightarrow \mathit{spaceTail}\ (y, m') \\
 & \qquad\qquad (\lambda(n', m'') \rightarrow \mathit{exec}'\ s\ (n + n', \mathit{free}\ 3\ m'')))
\end{array}
$$

Generalising $\mathit{exec}'\ s$ in both results to an arbitrary function c gives the following definition, which is now in continuation-passing style:

$$
\begin{array}{ll}
\textbf{type}\ \mathit{Con} & = (\mathit{Value}, \mathit{MemMng}) \rightarrow (\mathit{Value}, \mathit{MemMng}) \\
\mathit{spaceTail} & :: (\mathit{Expr}, \mathit{MemMng}) \rightarrow \mathit{Con} \rightarrow (\mathit{Value}, \mathit{MemMng}) \\
\mathit{spaceTail}\ (\mathit{Val}\ v, m)\ c & = c\ (v, \mathit{free}\ 1\ m) \\
\mathit{spaceTail}\ (\mathit{Add}\ x\ y, m)\ c & = \mathit{spaceTail}\ (x, \mathit{alloc}\ 1\ m)\ (\lambda(n, m') \rightarrow \\
 & \qquad \mathit{spaceTail}\ (y, m')\ (\lambda(n', m'') \rightarrow \\
 & \qquad c\ (n + n', \mathit{free}\ 3\ m'')))
\end{array}
$$

In turn, the behaviour of the *space* function can be modified accordingly:

$$space\ e = \textbf{let}\ (v, m) = spaceTail\ (e, [\,], alloc\ 2\ initial)\ \textbf{in}\ (v, free\ 2\ m)$$

Direct style

The space measuring evaluator can be transformed from continuation-passing style back to a direct style by starting from the following specification:

$$c\ (spaceEval\ (e, m)) = spaceTail\ (e, m)\ c$$

The equation expresses that the continuation applied to the new function *spaceEval* gives the same final result as supplying the continuation as an argument to *spaceTail*. The definition for *spaceEval* can now be calculated by induction on e:

Case : $e = Val\ v$

$$\begin{array}{ll} & c\ (spaceEval\ (Val\ v, m)) \\ = & \quad \{\ \text{specification}\ \} \\ & spaceTail\ (Val\ v, m) \\ = & \quad \{\ \text{definition of } spaceTail\ \} \\ & c\ (v, free\ 1\ m) \end{array}$$

Case : $e = Add\ x\ y$

$$\begin{array}{ll} & c\ (spaceEval\ (Add\ x\ y, m)) \\ = & \quad \{\ \text{specification}\ \} \\ & spaceTail\ (Add\ x\ y, m)\ c \\ = & \quad \{\ \text{definition of } spaceTail\ \} \\ & spaceTail\ (x, alloc\ 1\ m)\ (\lambda(n, m') \rightarrow spaceTail\ (y, m') \\ & \qquad\qquad (\lambda(n', m') \rightarrow c\ (n + n', free\ 3\ m''))) \\ = & \quad \{\ \text{inductive assumption}\ \} \\ & (\lambda(n, m') \rightarrow spaceTail\ (y, m')\ (\lambda(n', m') \rightarrow c\ (n + n', free\ 3\ m'')))\ \\ & \qquad (spaceEval\ (x, alloc\ 1\ m)) \end{array}$$

$$
\begin{array}{ll}
= & \quad \{ \text{ introduce let-binding } \} \\
& \quad \textbf{let } (n, m') = \mathit{spaceEval}\ (x, \mathit{alloc}\ 1\ m) \\
& \quad \textbf{in } \mathit{spaceTail}\ (y, m')\ (\lambda(n', m') \rightarrow c\ (n + n', \mathit{free}\ 3\ m'')) \\
= & \quad \{ \text{ inductive assumption } \} \\
& \quad \textbf{let } (n, m') = \mathit{spaceEval}\ (x, \mathit{alloc}\ 1\ m) \\
& \quad \textbf{in } (\lambda(n', m') \rightarrow c\ (n + n', \mathit{free}\ 3\ m''))\ (\mathit{spaceEval}\ (y, m')) \\
= & \quad \{ \text{ introduce let-binding } \} \\
& \quad \textbf{let } (n, m') = \mathit{spaceEval}\ (x, \mathit{alloc}\ 1\ m) \\
& \qquad\ (n', m') = \mathit{spaceEval}\ (y, m') \\
& \quad \textbf{in } c\ (n + n', \mathit{free}\ 3\ m'')
\end{array}
$$

The application of the continuation to both sides can then be removed using extensionality, to give the following definition of *spaceEval*:

$$
\begin{array}{lll}
\mathit{spaceEval} & :: (\mathit{Expr}, \mathit{MemMng}) \rightarrow (\mathit{Value}, \mathit{MemMng}) \\
\mathit{spaceEval}\ (\mathit{Val}\ v, m) & = (v, \mathit{free}\ 1\ m) \\
\mathit{spaceEval}\ (\mathit{Add}\ x\ y, m) & = \textbf{let } (n, m') = \mathit{spaceEval}\ (x, \mathit{alloc}\ 1\ m) \\
& \qquad\qquad\ (n', m'') = \mathit{spaceEval}\ (y, m') \\
& \qquad\ \textbf{in } (n + n', \mathit{free}\ 3\ m'')
\end{array}
$$

In turn, the definition of the *space* function now becomes:

$$\mathit{space}\ e = \textbf{let } (v, m) = \mathit{spaceEval}\ (e, \mathit{alloc}\ 2\ \mathit{initial})\ \textbf{in } (v, \mathit{free}\ 2\ m)$$

At this point, we can modify the evaluator so that it only measures space usage, because in this case we never require the result of the evaluation. The function *spaceAcc* can be calculated by applying the *snd* function to *spaceEval*:

$$
\begin{array}{lll}
\mathit{spaceAcc} & :: (\mathit{Expr}, \mathit{MemMng}) \rightarrow \mathit{MemMng} \\
\mathit{spaceAcc}\ (\mathit{Val}\ v, m) & = \mathit{free}\ 1\ m \\
\mathit{spaceAcc}\ (\mathit{Add}\ x\ y, m) & = \textbf{let } m' = \mathit{spaceAcc}\ (x, \mathit{alloc}\ 1\ m) \\
& \qquad\qquad\ m'' = \mathit{spaceAcc}\ (y, m') \\
& \qquad\ \textbf{in } \mathit{free}\ 3\ m''
\end{array}
$$

The *space* function itself now becomes:

$$space\ e = free\ 2\ (spaceAcc\ (e, alloc\ 2\ initial))$$

Finally, the definitions for the functions *spaceAcc* and *space* can now be simplified by rewriting them in compositional style:

$$\begin{aligned}
&spaceAcc' && :: Expr \to MemMng \to MemMng \\
&spaceAcc'\ (Val\ v) && = free\ 1 \\
&spaceAcc'\ (Add\ x\ y) && = free\ 3 \circ spaceAcc'\ y \circ spaceAcc'\ x \circ alloc\ 1 \\
&space\ e && = free\ 2 \circ spaceAcc'\ e \circ alloc\ 2
\end{aligned}$$

These definitions clearly show that the space requirement to evaluate an expression is given by first allocating two units of space (to pair the expression with the empty stack), then running the auxiliary *spaceAcc′* function, and finally freeing the two units of space for the empty stack and pair constructor at the end. If the expression is a value, the *spaceAcc′* function frees one unit of space, because the *Val* constructor is no longer required. In the addition case, a single unit is first allocated to store the right hand value, then the left hand branch is evaluated, followed by the right, and finally three units of space are freed, corresponding to no longer requiring the additional unit allocated, the *Add* constructor, and two integers being collapsed down to one. In this manner, we now have a clear and precise account of the space that is allocated and freed during the evaluation of an expression.

Remove accumulator

At present, the *spaceAcc′* function passes around a memory manager accumulator. This structure may be removed by first defining a function *update*, which takes two memory managers and combines them in sequence. The *update* function defines how the memory requirements of two sub-expressions x and y, captured in memory manager pairs m and n, can be combined in order to express the cost of first evaluating x, and then evaluating y. In this manner, we can express the fact that the allocation cost of n can use up any free memory from m. The *update* function is defined by first applying the allocation costs of the first memory manager argument to the second such argument, and then freeing:

$$\begin{aligned} &\mathit{update} && :: \mathit{MemMng} \to \mathit{MemMng} \to \mathit{MemMng} \\ &\mathit{update}\ (f, a) && = \mathit{free}\ f \circ \mathit{alloc}\ a \end{aligned}$$

In order to re-express the space requirements function using *update* we require the property that performing an *update* with the *initial* memory manager results in the following identity, which can easily be proved from the definitions:

$$\mathit{update}\ m\ \mathit{initial} = m \quad (5)$$

In order to remove the accumulator, we start from the following specification:

$$\mathit{update}\ (\mathit{space}'\ e)\ m = \mathit{spaceAcc}'\ e\ m$$

This equation expresses that executing $\mathit{spaceAcc}'$ with the starting memory manager m gives the same result as executing the new (non-accumulator) version space' and then updating the resulting memory manager with m. As usual, the definition of the space' function can be calculated by induction on e:

Case : $e = \mathit{Val}\ v$

$$\begin{aligned} & \mathit{update}\ (\mathit{space}'\ (\mathit{Val}\ n))\ m \\ = & \quad \{\ \text{specification}\ \} \\ & \mathit{spaceAcc}'\ (\mathit{Val}\ n)\ m \\ = & \quad \{\ \text{definition of } \mathit{spaceAcc}'\ \} \\ & \mathit{free}\ 1\ m \\ = & \quad \{\ \text{property 5}\ \} \\ & \mathit{update}\ (\mathit{free}\ 1\ \mathit{initial})\ m \end{aligned}$$

Case : $e = \mathit{Add}\ x\ y$

$$\begin{aligned} & \mathit{update}\ (\mathit{space}'\ (\mathit{Add}\ x\ y))\ m \\ = & \quad \{\ \text{specification}\ \} \\ & \mathit{spaceAcc}'\ (\mathit{Add}\ x\ y)\ m \\ = & \quad \{\ \text{definition of } \mathit{spaceAcc}'\ \} \end{aligned}$$

$$
\begin{aligned}
& (\mathit{free}\ 3 \circ \mathit{spaceAcc}'\ y \circ \mathit{spaceAcc}'\ x \circ \mathit{alloc}\ 1)\ m \\
= & \quad \{\ \text{inductive assumptions for } x \text{ and } y\ \} \\
& (\mathit{free}\ 3 \circ \mathit{update}\ (\mathit{space}'\ y) \circ \mathit{update}\ (\mathit{space}'\ x) \circ \mathit{alloc}\ 1)\ m \\
= & \quad \{\ \text{property 5}\ \} \\
& \mathit{update}\ ((\mathit{free}\ 3 \circ \mathit{update}\ (\mathit{space}'\ y) \circ \mathit{update}\ (\mathit{space}'\ x) \circ \mathit{alloc}\ 1)\ \mathit{initial})\ m
\end{aligned}
$$

Removing the *update* from both sides gives the following result:

$$
\begin{aligned}
\mathit{space}'\ (\mathit{Val}\ v) &= \mathit{free}\ 1\ \mathit{initial} \\
\mathit{space}'\ (\mathit{Add}\ x\ y) &= (\mathit{free}\ 3 \circ \mathit{update}\ (\mathit{space}'\ y) \circ \mathit{update}\ (\mathit{space}'\ x) \circ \mathit{alloc}\ 1)\ \mathit{initial} \\
\mathit{space}\ e &= (\mathit{free}\ 1 \circ \mathit{update}\ (\mathit{space}'\ e) \circ \mathit{alloc}\ 1)\ \mathit{initial}
\end{aligned}
$$

These definitions are more complicated than the analogous accumulator versions, but can be used as an intermediate step in order to produce a function that only measures the amount of space that is required to be allocated.

Allocation requirements

A further development may be made by observing that the free space available at the end of evaluation is equal to the size of the expression, minus the size of its resulting value, plus the additional space allocated during evaluation. This is because at the end of evaluation the only space currently occupied is equal to the size of the value, and therefore the space freed up will be equal to the original size of the expression being evaluated plus the additional space that was allocated during evaluation. This observation can be expressed by the following implication:

$$(f, a) = \mathit{space}'\ e \quad \Rightarrow \quad f = (\mathit{size}\ e - \mathit{size}\ (\mathit{eval}\ e)) + a$$

As stated earlier, we define the amount of space required to evaluate an expression as the allocated amount in the resulting memory manager pair. A direct requirements function, *req*, can be calculated from the specification

$$\mathit{req} = \mathit{allocated} \circ \mathit{space}$$

as follows:

$$
\begin{array}{ll}
 & \mathit{req}\ e \\
= & \quad \{\ \text{specification}\ \} \\
 & \mathit{allocated}\ (\mathit{space}\ e) \\
= & \quad \{\ \text{definition of } \mathit{space}\ \} \\
 & \mathit{allocated}\ ((\mathit{free}\ 1 \circ \mathit{update}\ (\mathit{space}'\ e) \circ \mathit{alloc}\ 1)\ \mathit{initial}) \\
= & \quad \{\ \text{definition of composition}\ \} \\
 & \mathbf{let}\ (f, a) = \mathit{space}'\ e\ \mathbf{in}\ (\mathit{allocated} \circ \mathit{free}\ 1 \circ \mathit{free}\ f \circ \mathit{alloc}\ a \circ \mathit{alloc}\ 1)\ \mathit{initial} \\
= & \quad \{\ \text{property 4}\ \} \\
 & \mathbf{let}\ (f, a) = \mathit{space}'\ e\ \mathbf{in}\ \mathit{allocated}\ (\mathit{alloc}\ a \circ \mathit{alloc}\ 1)\ \mathit{initial} \\
= & \quad \{\ \text{property 2}\ \} \\
 & \mathbf{let}\ (f, a) = \mathit{space}'\ e\ \mathbf{in}\ \mathit{allocated}\ (\mathit{alloc}\ (a + 1))\ \mathit{initial} \\
= & \quad \{\ \text{definition of } \mathit{alloc}\ \} \\
 & \mathbf{let}\ (f, a) = \mathit{space}'\ e\ \mathbf{in}\ \mathit{allocated}\ (0, a + 1) \\
= & \quad \{\ \text{definition of } \mathit{allocated}\ \} \\
 & \mathbf{let}\ (f, a) = \mathit{space}'\ e\ \mathbf{in}\ a + 1 \\
= & \quad \{\ \text{define } \mathit{req}'\ e = \mathit{allocated}\ (\mathit{space}'\ e)\ \} \\
 & \mathit{req}'\ e + 1
\end{array}
$$

In turn, a definition for the auxiliary function req' can be calculated by induction on the expression e using the specification $\mathit{req}'\ e = \mathit{allocated}\ (\mathit{space}'\ e)$:

Case : $e = \mathit{Val}\ v$

$$
\begin{array}{ll}
 & \mathit{req}'\ (\mathit{Val}\ v) \\
= & \quad \{\ \text{specification}\ \} \\
 & \mathit{allocated}\ (\mathit{space}'\ (\mathit{Val}\ v)) \\
= & \quad \{\ \text{definition of } \mathit{space}'\ \} \\
 & \mathit{allocated}\ (\mathit{free}\ 1\ \mathit{initial}) \\
= & \quad \{\ \text{definition of } \mathit{free}\ \} \\
 & \mathit{allocated}\ (0, 1) \\
= & \quad \{\ \text{definition of } \mathit{allocated}\ \} \\
 & 0
\end{array}
$$

Case : $e = Add\ x\ y$

$$
\begin{array}{ll}
 & req'\ (Add\ x\ y) \\
= & \quad \{\ \text{specification}\ \} \\
 & allocated\ (space'\ (Add\ x\ y)) \\
= & \quad \{\ \text{definition of } space'\ \} \\
 & allocated\ ((free\ 3 \circ update\ (space'\ y) \circ update\ (space'\ x) \circ alloc\ 1)\ initial) \\
= & \quad \{\ \text{definition of } update\ \} \\
 & \textbf{let}\ (f', a') = space'\ y \\
 & \textbf{in}\ allocated\ ((free\ (f' + 3) \circ alloc\ a' \circ update\ (space'\ x) \circ alloc\ 1)\ initial) \\
= & \quad \{\ \text{definition of } allocated\ \} \\
 & allocated\ ((alloc\ (allocated\ (space'\ y)) \circ update\ (space'\ x) \circ alloc\ 1)\ initial) \\
= & \quad \{\ \text{inductive assumption}\ \} \\
 & allocated\ ((alloc\ (req'\ y) \circ update\ (space'\ x) \circ alloc\ 1)\ initial) \\
= & \quad \{\ \text{definition of } update\ \} \\
 & \textbf{let}\ (f, a) = space'\ x\ \textbf{in}\ allocated\ ((alloc\ (req'\ y) \circ free\ f \circ alloc\ a \circ alloc\ 1)\ initial) \\
= & \quad \{\ \text{property 2}\ \} \\
 & \textbf{let}\ (f, a) = space'\ x\ \textbf{in}\ allocated\ ((alloc\ (req'\ y) \circ free\ f \circ alloc\ (a + 1))\ initial) \\
= & \quad \{\ \text{definition of } alloc \text{ and } free\ \} \\
 & \textbf{let}\ (f, a) = space'\ x\ \textbf{in}\ allocated\ (alloc\ (req'\ y)\ (f, a + 1)) \\
= & \quad \{\ \text{definition of } alloc\ \} \\
 & \textbf{let}\ (f, a) = space'\ x\ \textbf{in}\ allocated\ (f \dot{-} req'\ y, (a + 1) + (req'\ y \dot{-} f)) \\
= & \quad \{\ \text{definition of } allocated\ \} \\
 & \textbf{let}\ (f, a) = space'\ x\ \textbf{in}\ a + 1 + (req'\ y \dot{-} f) \\
= & \quad \{\ (f, a) = space'\ x \Rightarrow f = (size\ e - size\ (eval\ e)) + a\ \} \\
 & \textbf{let}\ (f, a) = space'\ x\ \textbf{in}\ a + 1 + (req'\ y \dot{-} (size\ x - 1 + a)) \\
= & \quad \{\ \text{inductive assumption}\ \} \\
 & \textbf{let}\ a = req'\ x\ \textbf{in}\ a + 1 + (req'\ y \dot{-} (size\ x - 1 + a)) \\
= & \quad \{\ \text{definition of } \dot{-}\ \} \\
 & \textbf{let}\ a = req'\ x\ \textbf{in}\ a + 1 + max\ (a' - ((size\ x - 1) + a))\ 0 \\
= & \quad \{\ \text{arithmetic}\ \}
\end{array}
$$

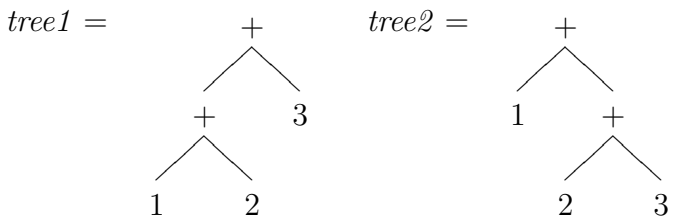
$$\textbf{let } a = req'\ x \textbf{ in } 1 + max\ a\ (a' - (size\ x - 1))$$

In conclusion, we have derived the following definitions:

$$\begin{aligned} req\ e &= 2 + req'\ e \\ req'\ (Val\ v) &= 0 \\ req'\ (Add\ x\ y) &= 1 + max\ (req'\ x)\ (req'\ y - (size\ x - 1)) \end{aligned}$$

This result is intuitive, because it describes how any space recovered from evaluating the left branch of an addition, consisting of both the space the expression was occupying and the additionally allocated space, can be re-used in evaluating the right-hand branch. The resulting function states that the requirements for evaluation are two plus the result of the auxiliary function req'. If the expression is a value then the result is zero, otherwise the requirements are one plus the maximum of the requirements of the left branch and the requirements of the right branch, but subtracting the size of the left branch minus one.

For example, evaluation of *tree1* below requires four units of space, whereas *tree2*, despite having the same number of nodes, requires one less unit because of the different branching structure. In particular, evaluation of *tree2* is able to reuse the space from evaluating the left branch in order to evaluate the larger right branch:



$req\ tree1 = 4$ $\qquad$ $req\ tree2 = 3$

As in the previous chapter, we would like to apply our approach to determine the space performance of the example list functions expressed using the extended language. The machine derived in section 3.4 can be instrumented with space information in a similar manner to that for the simple language of integers and addition. However, the space

performance results are not as straightforward as we may expect. In particular, the problem is that the environment in the machine is not threaded in the same way that the stack is threaded, but rather is duplicated.

For example, consider the case for addition:

$$eval\ (Add\ x\ y)\ env = eval\ x\ env + eval\ y\ env$$

The environment *env* is passed down both branches of the *Add*, in order to be able to look up the value of variables contained in the branches. The corresponding case in the abstract machine evaluates the left branch x, and saves the right branch y, along with the environment *env* and stack c, so that it can be evaluated later:

$$evalMach\ (Add\ x\ y)\ env\ c = evalMach\ x\ env\ (AddL\ c\ y\ env)$$

Instrumenting this transition with space usage gives the cost as $alloc\ (size\ env)$, which means it depends on the size of the current environment. This results in a space function that is difficult to reason about because it cannot be easily generalised; even reasoning about an addition is dependent on what is currently in the environment.

This problem could be resolved by making the environment threaded, so that it is passed down one branch, returned and then passed down another branch. In order to achieve this any additions to the environment made in one branch would need to be marked and then rolled back before being passed down the next branch. This would be possible in the current implementation as a list, because no mappings are forgotten and variables just become unreachable if the same variable name occurs earlier in the list.

Another approach would be to view the environment as global, in the sense that it is shared among all evaluations. However, this method would conflict with our approach to modeling deallocation, because any new additions to the environment would be kept until the end of evaluation. In order to address this issue, an additional garbage collection process would be required to remove redundant bindings from the environment: this process is known as environment-trimming [41].

Rather than further pursuing an environment based approach, at this point we return to our original examples of space performance given in section 1.4. These ex-

amples directly reflect a programmer's intuition that evaluation proceeds by a process of unfolding definitions and substituting for their arguments. Following the lead of these examples, we will now consider a substitution based approach to evaluation. However, substitution does incur a time overhead, because every function application requires traversing the function body to find occurrences of its arguments. It can also increase the space requirements, because the substituted expression may be duplicated. On the other hand, an environment model is not without its own overheads. In particular, for each of the proposed models of environments there is either a space overhead, such as duplicating or maintaining a global untrimmed environment, or a time overhead, for example in rolling back bindings or trimming. Our primary focus, however, is on exploring a calculational approach to obtaining space information, rather than finding the most efficient implementation.

5.5 Substitution evaluator

Our starting evaluator is a combination of the evaluation semantics given in section 3.4 and the substitution semantics presented in section 2.10. The difference in the language syntax is that the *Value* datatype no longer contains closures, because we no longer have environments:

$$\begin{aligned}&\textbf{data}\ \mathit{Expr} = \mathit{Var}\ \mathit{Int} \mid \mathit{Abs}\ \mathit{Int}\ \mathit{Expr} \mid \mathit{App}\ \mathit{Expr}\ \mathit{Expr}\\ &\qquad\qquad \mid \mathit{Add}\ \mathit{Expr}\ \mathit{Expr} \mid \mathit{Val}\ \mathit{Value}\\ &\qquad\qquad \mid \mathit{Nil} \mid \mathit{Cons}\ \mathit{Expr}\ \mathit{Expr}\\ &\qquad\qquad \mid \mathit{Foldr}\ \mathit{Expr}\ \mathit{Expr}\ \mathit{Expr}\\ &\textbf{data}\ \mathit{Value} = \mathit{Const}\ \mathit{Int}\\ &\qquad\qquad \mid \mathit{VCons}\ \mathit{Value}\ \mathit{Value} \mid \mathit{VNil}\end{aligned}$$

The type of the *eval* function is modified so that it has return type *Expr*, instead of *Value*, though the expression returned should either be an abstraction or a value tagged with a *Val* constructor. The evaluation semantics for each language construct is the same as in our environment evaluator, except that we substitute into the body of the abstraction when evaluating an application:

$$
\begin{array}{lll}
eval & :: & Expr \to Expr \\
eval\ (Abs\ x\ e) & = & Abs\ x\ e \\
eval\ (App\ f\ e) & = & \textbf{let}\ (Abs\ x\ e') = eval\ f \\
 & & \quad v = eval\ e \\
 & & \textbf{in}\ eval\ (subst\ e'\ v\ x) \\
eval\ (Val\ v) & = & Val\ v \\
eval\ (Add\ x\ y) & = & \textbf{let}\ (Val\ (Const\ m)) = eval\ x \\
 & & \quad (Val\ (Const\ n)) = eval\ y \\
 & & \textbf{in}\ (Val\ (Const\ (n + m))) \\
eval\ Nil & = & Val\ VNil \\
eval\ (Cons\ x\ xs) & = & \textbf{let}\ (Val\ v) = eval\ x \\
 & & \quad (Val\ vs) = eval\ xs \\
 & & \textbf{in}\ Val\ (VCons\ v\ vs) \\
eval\ (Foldr\ f\ v\ xs) & = & \textbf{case}\ eval\ xs\ \textbf{of} \\
\quad (Val\ VNil) & & \to eval\ v \\
\quad (Val\ (VCons\ y\ ys)) & \to & \textbf{let}\ f' = eval\ f \\
 & & \quad z = eval\ (Foldr\ f'\ v\ (Val\ ys)) \\
 & & \textbf{in}\ eval\ (App\ (App\ f'\ (Val\ y))\ z)
\end{array}
$$

For the purposes of defining a substitution function, we assume that all expressions have been α-converted to ensure that variable renaming is not required, as this would unnecessarily complicate the process of reasoning about space requirements:

$$
\begin{array}{lll}
subst & & :: Expr \to Expr \to Int \to Expr \\
subst\ (Var\ y)\ v\ x & \mid x == y & = v \\
 & \mid otherwise & = Var\ y \\
subst\ (App\ f\ e')\ v\ x & & = App\ (subst\ f\ v\ x)\ (subst\ e'\ v\ x) \\
subst\ (Abs\ y\ e)\ v\ x & \mid x == y & = Abs\ y\ e \\
 & \mid otherwise & = Abs\ y\ (subst\ e\ v\ x) \\
subst\ (Val\ v')\ v\ x & & = Val\ v' \\
subst\ (Add\ x'\ y)\ v\ x & & = Add\ (subst\ x'\ v\ x)\ (subst\ y\ v\ x) \\
subst\ Nil\ v\ x & & = Nil
\end{array}
$$

$$
\begin{array}{ll}
subst\ (Cons\ x'\ xs)\ v\ x & = Cons\ (subst\ x'\ v\ x)\ (subst\ xs\ v\ x) \\
subst\ (Foldr\ f\ v'\ xs)\ v\ x & = Foldr\ (subst\ f\ v\ x)\ (subst\ v'\ v\ x)\ (subst\ xs\ v\ x)
\end{array}
$$

Starting with this evaluator and substitution function, the same process is followed as in chapter 3 of first transforming the definitions into continuation-passing style. The new return type of *eval* allows us to thread the continuation (a function of type $Expr \rightarrow Expr$) through the substitution function and hence permit the cost of the substitution to be considered along with the cost of evaluation. Next, we defunctionalize the continuation to produce an abstract machine that uses substitution. The resulting continuation data structure requires more constructs than the environment machine, because of the propagation of the continuation through the machine version of the substitution function. For example, a construct with three arguments, $C\ a\ b\ c$, requires three constructors for performing substitution, one to substitute into each argument, saving the other arguments and continuation. The allocation instrumentation presented earlier can then be added to the abstract machine, to produce a machine that additionally measures its own space requirements. We omit the details of the resulting machine here for reasons of space, but they are given in appendix A.

Space function

Taking the instrumented abstract machine and inverting the calculational process as shown in chapter 4 results in the following space requirements function:

$$
\begin{array}{ll}
space & :: Expr \rightarrow MemMng \\
space\ e & = (free\ 1 \circ space'\ e \circ alloc\ 1)\ initial \\
space' & :: Expr \rightarrow MemMng \rightarrow MemMng \\
space'\ (Abs\ x\ e) & = id \\
space'\ (App\ f\ e) & = \textbf{let}\ (Abs\ x\ e') = eval\ f \\
 & \qquad v = eval\ e \\
 & \qquad e'' = subst\ e'\ v\ x \\
 & \quad \textbf{in}\ space'\ e'' \circ free\ 1 \circ subst'\ e'\ v\ x \circ free\ 1 \circ space'\ e \circ space'\ f \\
space'\ (Val\ v) & = \quad id
\end{array}
$$

$$
\begin{array}{lcl}
space'\ (Add\ x\ y) & = & free\ 2 \circ space'\ y \circ free\ 2 \circ space'\ x \\
space'\ Nil & = & alloc\ 1 \\
space'\ (Cons\ x\ xs) & = & space'\ xs \circ free\ 1 \circ space'\ x \\
space'\ (Foldr\ f\ v\ xs) = & & \textbf{case}\ eval\ xs\ \textbf{of} \\
\quad (Val\ VNil) & \rightarrow & space'\ v \circ free\ (3 + size\ f) \circ space'\ xs \\
\quad (Val\ (VCons\ y\ ys)) & \rightarrow & \textbf{let}\ f' = eval\ f \\
 & & \qquad z = eval\ (Foldr\ f\ v\ (Val\ ys)) \\
 & & \qquad\ \textbf{in}\ space'\ (App\ (App\ f\ (Val\ y))\ z) \circ \\
 & & \quad alloc\ 2 \circ space'\ (Foldr\ f\ v\ (Val\ ys)) \circ \\
 & & \quad alloc\ (2 + size\ f') \circ space'\ f \circ free\ 2 \circ space'\ xs
\end{array}
$$

The corresponding substitution function that measures space requirements is:

$$
\begin{array}{lll}
subst' & & :: Expr \rightarrow Expr \rightarrow Int \rightarrow MemMng \rightarrow MemMng \\
subst'\ (Var\ y)\ v\ x & \mid x == y & = free\ 3 \\
 & \mid otherwise & = free\ (1 + size\ v) \\
subst'\ (App\ f\ e')\ v\ x & & = subst'\ e'\ v\ x \circ subst'\ f\ v\ x \circ alloc\ (1 + size\ v) \\
subst'\ (Abs\ y\ e)\ v\ x & \mid x == y & = free\ (1 + size\ v) \\
 & \mid otherwise & = subst'\ e\ v\ x \\
subst'\ (Val\ v')\ v\ x & & = free\ (1 + size\ v) \\
subst'\ (Add\ x'\ y)\ v\ x & & = subst'\ y\ v\ x \circ subst'\ x'\ v\ x \circ alloc\ (1 + size\ v) \\
subst'\ Nil\ v\ x & & = free\ (1 + size\ v) \\
subst'\ (Cons\ x'\ xs)\ v\ x & & = subst'\ x'\ v\ x \circ subst'\ xs\ v\ x \circ alloc\ (1 + size\ v) \\
subst'\ (Foldr\ f\ v'\ xs)\ v\ x & & = subst'\ xs\ v\ x \circ subst'\ v'\ v\ x \circ alloc\ (1 + size\ v) \circ \\
 & & \quad subst'\ f\ v\ x \circ alloc\ (1 + size\ v)
\end{array}
$$

5.6 Examples

We can now use the derived *space* function to consider the space requirements of the example functions presented in chapter 4.

Sum

To measure the space performance of the *sum* function, defined in section 4.3 using fold-right, the *space* function is specialised to *sum* applied to a list of integers *xs*, the result simplified using the memory manager properties, and finally expressed as a closed-form function over the length of the input list. The result is that the allocation requirements for evaluating *sum xs* is $1 + 9 * \mathit{length}\ \mathit{xs}$.

The *space* function also returns the amount of free space at the end of the evaluation. As discussed earlier the relationship between the allocated and freed amount is that the size of the initial data structure minus the size of the result plus the additional allocation amount is equal to the freed amount. This relationship between the allocation and freed amounts provides one way to check that no errors have been introduced in the calculation process. In the case of *sum* the size of the initial data structure is the size of the input list *xs* combined with the initial *Top* constructor, giving $3 + 3 * \mathit{length}\ \mathit{xs}$, whilst the size of the result, which will be of the form $\mathit{Val}\ (\mathit{Const}\ n)$, is 3. The property is maintained in the case of *sum*, since the freed amount $1 + 12 * \mathit{length}\ \mathit{xs}$ is equal to $((3 + 3 * \mathit{length}\ \mathit{xs}) - 3) + (1 + 9 * \mathit{length}\ \mathit{xs})$.

Sum with an accumulator

In section 4.3 we considered the time requirements of summing a list using an accumulator. We expect an accumulator version of *sum* to require less additional space, because additions can be performed without having to expand the whole list first, so we should only require the additional space to hold the accumulator.

The first implementation of this idea, *sumAcc*, uses fold-right to generate a function which was then applied to an initial accumulator value of zero. Applying the *space* function to this definition, does not however give the expected saving in space requirements. In particular, the allocated space for evaluating *sumAcc xs* is $1 + 24 * \mathit{length}\ \mathit{xs}$, which is considerably worse than the non-accumulating version, which had an overhead of 9 units of space per list element. This shows that although

extensionally a fold-left function can be re-expressed using fold-right, this does not mean that the space requirements of the two functions will be the same.

If the language and the *space* function is extended with fold-left, as defined in section 4.3, then we do indeed obtain the desired constant space requirements, namely 1 unit if the input list is empty, and 14 units for any non-empty list.

Reverse

To analyse the space performance of the *reverse* function, defined using fold-right as in section 4.3, it again helps to first consider the requirements of *append*. The *append* function is given by folding the list constructor over the first argument list, and in the empty list case returning the second argument list. Unsurprisingly, *append xs ys* has the same allocated space requirements as the *sum* function, because they both have the same structure, but with *Add* replaced by *Cons*. However, the freed amount at the end of evaluation is less than for the *sum* function, at $12+9*\mathit{length}\ \mathit{xs}$, because the result list is only slightly smaller than the size of the argument lists added together; in particular, there is one less empty list constructor.

Using the space requirements of *append*, the allocation requirements of evaluating *reverse* applied to a list of integers *xs* is $1 + 31 * \mathit{length}\ \mathit{xs}$.

Fast reverse

As considered in section 4.3, the quadratic time requirements of *reverse* may be improved by redefining the function using an accumulator. The first accumulator version, *fastrev*, was defined using fold-right and we showed that this version improved the performance to linear time. It would also be useful to know how this definition affects the space requirements. Applying the *space* function to *fastrev xs* returns the allocated space requirements of $24 * \mathit{length}\ \mathit{xs} - 23$. Hence the accumulator version also improves the space performance by 23%, though it still has linear requirements.

Re-defining the definition of *reverse* using fold-left instead of fold-right further improves the space performance because the allocation requirements become a con-

stant factor: if the input list is empty then the allocation requirements are simply 1 unit, otherwise they are a constant 13 units.

Average

Our final example concerns the *average* function, as given in section 4.3. For the purposes of this example we first add a division operator, pairs and projection functions in the language, using the semantics given in the previous chapter, and then apply the same calculational process to measure the space requirements of these constructs.

The first version of *average* calculates the length and sum as separate traversals of the argument list and then produces the average. Analysing the space requirements of this function gives the allocation requirements of *average xs* as $9 + 9 * \mathit{length}\ \mathit{xs}$. This amount is less than may be expected, considering that evaluation of the *sum xs* has space requirements of $1 + 9 * \mathit{length}\ \mathit{xs}$, because the space required to evaluate the length of the list can be reclaimed to calculate the sum.

Re-expressing the definition of *average* by exploiting the banana-split property of fold-right results in a definition that calculates the length and sum of the list in one pass. The space requirements to evaluate this version applied to an argument list *xs* are $1 + 18 * \mathit{length}\ \mathit{xs}$ units of additional space. The space requirements are now greater because we cannot re-use the space for each list traversal.

An alternative definition for *average* using fold-left utilises an accumulator to calculate both the length and sum in the same way, the only difference being the order of the arguments in the function to the fold, as shown below:

$$\begin{aligned}&\mathit{averagel}\ \mathit{xs} = (\lambda a \rightarrow \mathit{fst}\ a\ `\mathit{div}`\ \mathit{snd}\ a)\\&\quad (\mathit{foldl}\ (\lambda a\ x \rightarrow (x + \mathit{fst}\ a, 1 + \mathit{snd}\ a))\ (0,0)\ \mathit{xs})\end{aligned}$$

The fold-left definition has, once again, improved space performance compared to the fold-right version. The allocation requirements for this definition are constant: calculating the average of an empty list (which amounts to determining that the result in this case is undefined) requires one unit of additional space, while calculating the average of a non-empty list requires 13 units of additional space for evaluation.

5.7 Benchmarking

Similarly to our approach for time in the previous chapter, we now put our results in context through benchmarking against Hugs and GHC.

5.7.1 Cell and byte allocations

As well as printing the number of reductions required to evaluate an expression, Hugs can also output the number of cells that were allocated, which may be used as a rough estimate of how much memory was required to perform the evaluation. More precise results can be obtained from the GHC profiler, which outputs the number of bytes that were allocated during execution. In both Hugs and GHC, the space information that is provided will also include the space required for the arguments to the function being benchmarked. In contrast to our approach to time benchmarking in the previous chapter, it proved more difficult to separate out this additional cost. For this reason, when considering the space requirements of a function, we add the size of the arguments to the results derived from our theory.

For each of our example functions, the two tables below compare the results of applying our theory with the number of cells allocated in Hugs, and bytes allocated in GHC. For each function in column one, column two displays the result of applying our theory (plus the additional cost for the list argument), column three gives the space requirements using our original evaluation function, and column four shows the cost when the example function is implemented directly in Haskell.

Hugs

Function	Theory (*steps*)	Cells (*eval*)	Cells (Haskell)
sum	$3 + 12n$	$128_{42.7} + 91_{7.6}n$	$35_{11.7} + 2_{0.17}n$
sumAcc	$3 + 27n$	$184_{61.3} + 161_{6.0}n$	$36_{12} + 5_{0.19}n$
suml	$16 + 3n$	$1288 + 91_{30.3}n$	$35_{2.25} + 3_{1.0}n$
reverse	$3 + 34n$	$52_{17.3} + 395_{11.6}n + \frac{83(n-1)n}{2}$	$38_{12.7} + 18_{0.53}n + \frac{3(n-1)n}{2}$
fastrev	$-21 + 27n$	$177_{-8.4} + 270_{10.0}n$	$38_{-1.8} + 18_{0.67}n + \frac{3(n-1)n}{2}$
reversel	$15 + 3n$	$105_{7.0} + 200_{66.7}n$	$38_{2.5} + 17_{5.7}n$
average	$7 + 12n$	$293_{41.9} + 179_{14.9}n$	$115_{16.4} + 12_{1}n$
averageFused	$3 + 21n$	$252_{84.0} + 191_{9.1}n$	$112_{37.3} + 21\ 7.0n$
averagel	$15 + 3n$	$249_{37.3} + 191_{63.7}n$	$112_{7.5} + 22_{7.3}n$

GHC

Function	Theory(*steps*)	Bytes (*eval*)	Bytes (Haskell)
sum	$3 + 12n$	$3870_{1023.3} + 652_{54.3}n$	$2400_{800} + 64_{5.3}n$
sumAcc	$3 + 27n$	$4540_{1513.3} + 1072_{39.7}n$	$2400_{800} + 84_{3.1}n$
suml	$16 + 3n$	$3888_{243} + 648_{216}n$	$2400_{150} + 68_{22.6}n$
reverse	$3 + 34n$	$3084_{1028} + 3419_{100.6}n$	$328_{109.3} + 203_{6.0}n$
		$+264n(n-1)$	$+14n(n-1)$
fastrev	$-21 + 27n$	$3332_{-158.7} + 2195_{81.3}n$	$48_{2.3} + 179_{6.6}n$
reversel	$15 + 3n$	$2208_{147.2} + 1771_{590.3}n$	$20_{1.3} + 163_{54.3}n$
average	$7 + 12n$	$5304_{771.9} + 1232_{102.7}n$	$2552_{364.6} + 88_{7.3}n$
averageFused	$3 + 21n$	$4924_{1641.3} + 1208_{57.5}n$	$2508_{836.0} + 124_{5.9}n$
averagel	$15 + 3n$	$4920_{328} + 1204_{401.3}n$	$2500_{166.7} + 128_{42.7}n$

In each table, n is the length of the argument list, and the numeric subscripts are not part of the formulae themselves, but provide an indication of the fit between our theory and the number of cells in Hugs and bytes in GHC. For example, the subscript of 42.7 in the first row of the Hugs table indicates that there are 42.7 initial cells for

each initial unit in our theory, and is calculated by dividing 128 by 3.

In contrast to the time results in the previous chapter, the results of the space benchmarking do not correlate so consistently with our theory. This discrepancy may be accounted for by differing approaches to garbage collection. In particular, our theory models a continuous garbage collection policy, in which any redundant space is immediately collected and re-used after each step, and therefore our space requirements correspond to maximum residency. For practical reasons Hugs and GHC only invoke garbage collection periodically, and hence the space requirements produced from these systems are likely to include additional structures that could have been garbage collected and re-used. Our benchmarking results do, however, correlate to some degree with our theory in terms of which is the most space efficient version of each function, and similarly for the least efficient.

In our theory the least space-efficient definition for summing a list is the accumulator version using fold-right. This is true in Hugs and GHC, for both calling the *eval* function, and for the direct Haskell implementations. The most space efficient definition in the theory is the fold-left definition, however in Hugs and GHC this version is marginally worse than the non-accumulating fold-right definition.

For the different definitions of reverse, the fold-left version is the most space efficient in our theory, and the fold-right non-accumulating version the least space efficient. This is true in both the benchmarked results for Hugs and GHC.

The most space efficient definition of the average function in our theory is the fold-left fused version, and the least efficient was the non-fused definition that requires two separate passes over the list to compute the average. In Hugs this is reversed, in that the non-fused definition has the best space performance, and the fused fold-left definition that uses an accumulator to calculate the average in one pass, has the worst space performance. This result is confirmed in the GHC profiler with the direct Haskell implementations in GHC. However, the results from calling *eval* in our embedded language correlate with our theory, in that the fold-left definition is the more space efficient definition, and the non-fused definition is the worst. With the direct Haskell implementation, the results are the opposite, which is likely to arise

because GHC has an optimised implementation of tupling.

5.8 Summary

This chapter has shown how an abstract machine can be instrumented with space information, in order to calculate a high-level function that measures space requirements. However, this instrumentation process was more complicated than for time in the previous chapter. In particular, we required an approach that would allow the low-level structures of an abstract machine to be reflected in a high-level function that captures space requirements. The resulting solution used a memory manager to keep track of not only the amount of allocations required, but also the space deallocated, in order that this could be re-used later on if required.

To illustrate the use of our technique we considered a number of example functions that operate on lists. Once again, being able to derive precise formulae, rather than asymptotic bounds, allowed us to observe the effect of different optimisation techniques, such as accumulation and fusion. The chapter concluded by benchmarking using Hugs and GHC, and discussed the relationship between the results predicted by our theory and those produced by these systems. The next chapter will present an extended case study that utilises our approach to space analysis.

Chapter 6

Case study: compact fusion

In this chapter we apply the approach to reasoning about space requirements described in the previous chapter to an extended case study. In particular, we consider the requirements of a particular form of programs, namely those than can be expressed as so-called *hylomorphisms*.

6.1 Introduction

Hylomorphisms [57] represent a common programming pattern of using an intermediate data structure that is first built and then collapsed. More formally, a hylomorphism is the composition of an unfold and a fold: the unfold uses a seed value to generate a data structure, and the fold takes this structure and collapses it in some way. The space efficiency of such a composition may be improved by applying fusion techniques to eliminate the intermediate data structure. However, whether the space performance is *actually* improved depends on the fold being able to consume elements as they are generated. If this is not the case, then the result is the creation of the whole structure before any folding evaluation can take place, and the intermediate structure still effectively exists in the fused function.

In this chapter we explore this problem for the case when the intermediate structure is a list, and show how using an accumulating fold, *fold-left*, will improve the space performance. We then formalise the space behaviour of our solution using the

techniques developed in the previous chapter. The work in this chapter is based upon [58], but is presented in an extended form here.

6.2 Hylomorphisms

We will consider hylomorphisms where the intermediate data structure is a list; that is, the unfold generates a list from a seed, and the fold then consumes this list.

Unfold

The unfold function builds a list from an initial seed value. It takes three additional arguments: a predicate, p, to determine when to stop generating list elements, and two other functions, hd and tl, to produce the head of the list and to modify the seed value to pass to the recursive call and generate the rest of the list:

$$\begin{aligned}&\mathit{unfold} && :: (a \to \mathit{Bool}) \to (a \to b) \to (a \to a) \to a \to [b]\\ &\mathit{unfold}\ p\ hd\ tl = f\\ &\quad \textbf{where}\ f\ x = \textbf{if}\ p\ x\ \textbf{then}\ [\,]\ \textbf{else}\ hd\ x : f\ (tl\ x)\end{aligned}$$

In this manner, *unfold* has the following behaviour:

$$\mathit{unfold}\ p\ hd\ tl\ x = [hd\ x, hd\ (tl\ x), hd\ (tl\ (tl\ x)), ...]$$

For example, we can define a function *downFrom* using *unfold*, which takes a natural number n and produces a list of all the numbers from n down to 1, where *id* and *pred* are the identity and predecessor functions:

$$\mathit{downFrom} = \mathit{unfold}\ (== 0)\ \mathit{id}\ \mathit{pred}$$

Applying *downFrom* to the number 3 produces evaluation trace A in figure 6.1. We can use the shape of the trace to informally measure the amount of space required to evaluate the expression. If we assume that space may be re-used at each step of the evaluation, the space requirements may be taken as the maximum expression size generated during the evaluation, where the expression size can be estimated by counting constructor symbols. As we can see in the trace, the expression size

reaches its maximum when the list has been completely generated, producing a list of length equal to the argument to *downFrom*. Evaluating *downFrom* therefore requires additional space proportional to the value of its numeric argument, and so has linear space requirements.

Fold-right

The standard fold operator for lists takes two arguments, a binary operator $\oplus$ and a value v, and replaces every list constructor : with $\oplus$ and the empty list $[\,]$ by v:

$$\begin{aligned}
&\mathit{foldr} && :: (b \to c \to c) \to c \to [b] \to c \\
&\mathit{foldr}\ (\oplus)\ v = g \\
&\quad \textbf{where}\ g\ [\,] && = v \\
&\qquad\quad g\ (x : xs) && = x \oplus g\ xs
\end{aligned}$$

For example, a list $[a, b, c]$ would be folded as follows:

$$\mathit{foldr}\ (\oplus)\ v\ [a, b, c] = a \oplus (b \oplus (c \oplus v))$$

Calculating the product of a list of numbers can be expressed by folding the $*$ operator over the list, and substituting 1 in the empty list case:

$$\mathit{product} = \mathit{foldr}\ (*)\ 1$$

Applying *product* to the list $[3, 2, 1]$ gives the evaluation trace B shown in figure 6.1, and takes space proportional to the length of the list. This fold is called fold-right because, as shown in the trace, after replacing each list constructor : with the multiplication operator $*$, the resulting expression brackets to the right.

Hylomorphisms

A hylomorphism is the composition of an unfold with a fold, defined as follows:

$$\mathit{hylor} :: (a \to \mathit{Bool}) \to (a \to b) \to (a \to a) \to (b \to c \to c) \to c \to a \to c$$

$$\mathit{hylor}\ p\ hd\ tl\ (\oplus)\ v = \mathit{foldr}\ (\oplus)\ v \circ \mathit{unfold}\ p\ hd\ tl$$

FIGURE 6.1: Evaluation traces for *downFrom* and *product*

A)		B)	
	$downFrom\ 3$		$product\ (3:2:1:[\,])$
$=$	$f\ 3$	$=$	$g\ (3:2:1:[\,])$
$=$	$3:f\ 2$	$=$	$3 * g\ (2:1:[\,])$
$=$	$3:2:f\ 1$	$=$	$3 * (2 * g\ (1:[\,]))$
$=$	$3:2:1:f\ 0$	$=$	$3 * (2 * (1 * g\ [\,]))$
$=$	$3:2:1:[\,]$	$=$	$3 * (2 * (1 * 1))$
		$=$	$3 * (2 * 1)$
		$=$	$3 * 2$
		$=$	6

We use the name *hylor* for this function, rather than the standard *hylo*, to emphasise that it is specified in terms of fold-right. Within the definition for *hylor*, a list is generated by the unfold function and passed to the fold, which consumes it. However, the well-known hylo theorem [57] states that the two functions may be fused together to eliminate this intermediate data structure, and is expressed as follows:

$$hylor\ p\ hd\ tl\ (\oplus)\ v = h$$
$$\quad \textbf{where}\ h\ x = \textbf{if}\ p\ x\ \textbf{then}\ v\ \textbf{else}\ hd\ x \oplus h\ (tl\ x)$$

Note in particular that the function h does not construct and process an intermediate list, but produces the result value directly, without the need for such an intermediate structure. Now we will look at an example hylomorphism and see how the space performance is affected by applying the theorem.

Example: factorial

The factorial of a natural number n can be calculated by taking the product of the list of numbers from n down to 1. We can therefore express the factorial function as the composition of the two functions *product* and *downFrom*:

$$fact :: Int \to Int$$
$$fact = product \circ downFrom$$

This composition is a hylomorphism, because *downFrom* can be defined as an unfold and *product* can be defined as a fold, and so we can apply the *hylor* theorem (and inline the *pred* function) to give the following fused program:

$$
\begin{aligned}
&\mathit{fact} = h \\
&\quad \textbf{where } h\ x = \textbf{if } x == 0 \textbf{ then } 1 \textbf{ else } x * h\ (x-1)
\end{aligned}
$$

The purpose of fusing the program is to eliminate the creation of the intermediate list. In this case the argument and result are both integers, but a list is built in the process, so potentially we could perform the multiplication after each element of the list is generated and achieve evaluation in a constant amount of space. However, unwinding of the fused definition of factorial, given in trace A of figure 6.2, shows that this is not in fact the case. In particular, the trace shows that all of the list elements must be generated before any multiplications can occur. Although there is not an explicit list, the underlying structure is still there, with the list constructor replaced by the multiplication operator. Multiplication can only occur once the unfold has finished producing list elements, and the structure is then collapsed from the right.

The maximum expression size produced in the factorial example occurs when the list has been completely generated. Therefore, the amount of space required to evaluate the factorial of a natural number n is directly proportional to the value of n, and hence the fused program still requires linear space.

Impedance mismatch

The problem with this example is an impedance mismatch between unfold and fold-right; the former generates the list elements in left-to-right order, but the latter consumes them in right-to-left order. We take the term "impedance mismatch" from electrical engineering, where it is used to describe an electrical system that cannot efficiently accommodate the output from another system. The *hylor* theorem eliminates the overhead of constructing and destructing the intermediate list, but retains the underlying impedance mismatch and hence gives poor space performance.

FIGURE 6.2: Evaluation traces for *fact* and *factl*

A)		B)	
	$fact\ 3$		$factl\ 3$
$=$	$h\ 3$	$=$	$h\ 1\ 3$
$=$	$3 * (h\ 2)$	$=$	$h\ (1 * 3)\ 2$
$=$	$3 * (2 * (h\ 1))$	$=$	$h\ 3\ 2$
$=$	$3 * (2 * (1 * (h\ 0)))$	$=$	$h\ (3 * 2)\ 1$
$=$	$3 * (2 * (1 * 1))$	$=$	$h\ 6\ 1$
$=$	$3 * (2 * 1)$	$=$	$h\ (6 * 1)\ 0$
$=$	$3 * 2$	$=$	$h\ 6\ 0$
$=$	6	$=$	6

Fold-left

An alternative way to fold a list is to bracket the operator from the left, using a function *foldl* that behaves in the following manner:

$$foldl\ (\oplus)\ v\ [a, b, c] = (((v \oplus a) \oplus b) \oplus c)$$

The *foldl* function itself is defined using an accumulator that is simply returned in the empty list case, and combined with the head of the list to provide a new accumulator to process the tail of the argument list otherwise:

$$\begin{aligned} &foldl && :: (b \to a \to b) \to b \to [a] \to b \\ &foldl\ (\oplus)\ v = g\ v \\ &\quad \textbf{where}\ g\ a\ [\,] && = a \\ &\qquad\qquad g\ a\ (x : xs) && = g\ (a \oplus x)\ xs \end{aligned}$$

Duality

A well-known duality property [4] states that if an operator $\oplus$ is associative and has the value e as its unit, *foldr* and *foldl* called with these arguments will always give the same result. The converse result also holds, giving the following equivalence:

$$\begin{aligned} &x \oplus (y \oplus z) = (x \oplus y) \oplus z \\ &x \oplus e = e \oplus x = x \end{aligned} \quad \Leftrightarrow \quad foldr\ (\oplus)\ e = foldl\ (\oplus)\ e$$

In the case of the product function, $*$ is associative and has 1 as its unit, and hence this function can also be defined using *foldl*:

$$productl = foldl\ (*)\ 1$$

The evaluation trace A in figure 6.3 shows the result of evaluating $productl\ [3,2,1]$. In the third line of the trace, given by the expression $g\ (1*3)\ (2:1:[\,])$, we might wish to evaluate the $*$ first, before expanding the recursive call to g. Under Haskell's lazy evaluation strategy, however, the outermost redex is evaluated first, so the recursive call to g is evaluated before the multiplication. To force evaluation of the accumulator expression to occur first we can utilise strict application, denoted \$!, which forces evaluation of its second argument to head-normal form before the function argument is applied. Fold-left can be redefined using strict application as:

$$\begin{aligned} &foldl'\ (\oplus)\ v = g\ v \\ &\quad \textbf{where}\ g\ a\ [\,] \quad\ = a \\ &\qquad\quad g\ a\ (x:xs) = (g\ \$!\ (a \oplus x))\ xs \end{aligned}$$

Re-expressing product using $foldl'$ results in evaluation trace B in figure 6.3, in which a multiplication is now performed before each recursive call:

FIGURE 6.3: Evaluation traces for *productl* and $productl'$

A)		B)	
	$productl\ (3:2:1:[\,])$		$productl'\ (3:2:1:[\,])$
$=$	$g\ 1\ (3:2:1:[\,])$	$=$	$g\ 1\ (3:2:1:[\,])$
$=$	$g\ (1*3)\ (2:1:[\,])$	$=$	$g\ (1*3)\ (2:1:[\,])$
$=$	$g\ ((1*3)*2)\ (1:[\,])$	$=$	$g\ 3\ (2:1:[\,])$
$=$	$g\ (((1*3)*2)*1)\ [\,]$	$=$	$g\ (3*2)\ (1:[\,])$
$=$	$((1*3)*2)*1$	$=$	$g\ 6\ (1:[\,])$
$=$	$(3*2)*1$	$=$	$g\ (6*1)\ [\,]$
$=$	$6*1$	$=$	$g\ 6\ [\,]$
$=$	6	$=$	6

Left hylomorphism

By analogy with the established notion of the right-hylomorphism, we introduce the notion of a left-hylomorphism as the composition of an *unfold* with a *foldl′*:

$$hylol :: (a \to Bool) \to (a \to b) \to (a \to a) \to (c \to b \to c) \to c \to a \to c$$
$$hylol\ p\ hd\ tl\ (\oplus)\ v = foldl'\ ((\oplus))\ v \circ unfold\ p\ hd\ tl$$

The corresponding left-hylomorphism theorem states, once again, that the two functions may be fused together to eliminate the intermediate data structure:

$$hylol\ p\ hd\ tl\ (\oplus)\ v = h\ v$$
$$\textbf{where}\ h\ a\ x = \textbf{if}\ p\ x\ \textbf{then}\ a\ \textbf{else}\ (h\ \$!\ (a \oplus hd\ x))\ (tl\ x)$$

Although straightforward, to the best of our knowledge, this operator and its corresponding fusion theorem have not been considered before.

Proof of left hylomorphism theorem

Unfortunately, the standard technique of structural induction cannot be used to prove the left-hylomorphism theorem, because there is nothing to perform induction over, as we do not know the structure of the seed value. There is also no structured result to facilitate co-induction over. However, because both *foldl′* and *unfold* can be defined as least fixpoints, and therefore *hylol* is the composition of two such fixpoints, we can apply the "total fusion" [53] theorem, given by the following rule:

$$\frac{f\ a \circ g\ b = h\ (a \circ b)}{\mu f \circ \mu g = \mu h}$$

We can prove the total fusion theorem using fixpoint induction [59]:

$$\frac{P \bot \qquad \forall x. P\ x \Rightarrow P\ (f\ x)}{P\ (\mu f)}$$

This rule states that to prove a property P of a fixpoint μf, it is sufficient to show that P holds for $\bot$, and is preserved by f. The necessary assumptions are that types are complete partial orders (CPOs), which are sets with a partial-ordering $\sqsubseteq$, a least element $\bot$, and limits of all non-empty chains, and that programs are continuous functions, which are functions between CPOs that preserve the partial-order and limit structure. We also assume that P is admissible, in that it preserves limits of chains. Using fixpoint induction, the total fusion theorem can be proved as follows:

$$
\begin{array}{ll}
 & \mu f \circ \mu g = \mu h \\
\Leftrightarrow & \quad \{ \text{ define } P\ (a,b,c) = a \circ b = c \ \} \\
 & P\ (\mu f, \mu g, \mu h) \\
\Leftarrow & \quad \{ \text{ fixpoint induction } \} \\
 & P\ (\bot,\bot,\bot) \quad \wedge \quad \forall a,b,c.P\ (a,b,c) \Rightarrow P\ (f\ a, g\ b, h\ c)
\end{array}
$$

Showing that the first conjunct is satisfied is trivial, because $\bot \circ \bot = \bot$, so we proceed straight to verifying the second conjunct:

$$
\begin{array}{ll}
 & P\ (f\ a, g\ b, h\ c) \\
\Leftrightarrow & \quad \{ \text{ definition of } P \ \} \\
 & f\ a \circ g\ b = h\ c \\
\Leftarrow & \quad \{ \text{ assumption: } P\ (a,b,c), \text{ i.e. } a \circ b = c \ \} \\
 & f\ a \circ g\ b = h\ (a \circ b)
\end{array}
$$

This completes the proof, apart from showing that the predicate P is admissible, which is immediate from the fact that any equality between continuous functions can be shown to be admissible, and that composition preserves continuity.

To apply total fusion in the case of the left-hylomorphism theorem, first we need to re-express *unfold*, *foldl′* and *hylol* in terms of least fixpoints:

$$
\begin{array}{ll}
\mathit{unfold}\mu\ p\ \mathit{hd}\ \mathit{tl} & = \mu\ \mathit{unfold}' \\
\quad \textbf{where}\ \mathit{unfold}'\ g & = \lambda x \rightarrow \textbf{if}\ p\ x\ \textbf{then}\ [\,]\ \textbf{else}\ \mathit{hd}\ x : g\ (\mathit{tl}\ x)
\end{array}
$$

$$
\begin{aligned}
&\mathit{foldl'}\mu\ (\oplus)\ a &&= \mu\ \mathit{foldl''}\\
&\quad \textbf{where}\ \mathit{foldl''}\ g &&= \lambda xs\ a \rightarrow \textbf{case}\ xs\ \textbf{of}\\
&&&\qquad [\,] \rightarrow a\\
&&&\qquad y : ys \rightarrow g\ ys\ \$!\ (a \oplus y)\\
&\mathit{hylol}\mu\ p\ hd\ tl\ (\oplus)\ v &&= \mu\ \mathit{hylol'}\\
&\quad \textbf{where}\ \mathit{hylol'}\ h &&= \lambda x\ a \rightarrow\ \textbf{if}\ p\ x\ \textbf{then}\ a\ \textbf{else}\ h\ (tl\ x)\ (a \oplus hd\ x)
\end{aligned}
$$

Note that the list and accumulator arguments have been swapped in the *foldl''* and *hylol'* functions, in order that the list is now the first argument. The rationale for this approach is to make it easier to compose the fold-left and unfold in the proof, in that the result of the unfold (a list) is now the first argument to the fold-left.

We can now prove the left-hylomorphism theorem:

$$
\begin{aligned}
&\mathit{foldl'}\ (\oplus)\ v \circ \mathit{unfold}\ p\ hd\ tl = \mathit{hylol}\ p\ hd\ tl\ (\oplus)\ v\\
\Leftrightarrow\quad &\{\ \text{definition of the functions}\ \}\\
&\mu\ \mathit{foldl''} \circ \mu\ \mathit{unfold'} = \mu\ \mathit{hylol'}\\
\Leftarrow\quad &\{\ \text{total fusion}\ \}\\
&\mathit{foldl''}\ b \circ \mathit{unfold'}\ c = \mathit{hylol'}\ (b \circ c)
\end{aligned}
$$

The final equation can be verified as follows:

$$
\begin{aligned}
&(\mathit{foldl''}\ b \circ \mathit{unfold'}\ c)\ x\ a\\
=\quad &\{\ \text{definition of}\ \circ\ \}\\
&\mathit{foldl''}\ b\ (\mathit{unfold'}\ c\ x)\ a\\
=\quad &\{\ \text{definition of}\ \mathit{unfold'}\ \}\\
&\mathit{foldl''}\ b\ (\textbf{if}\ p\ x\ \textbf{then}\ [\,]\ \textbf{else}\ hd\ x : c\ (tl\ x))\ a\\
=\quad &\{\ \text{definition of}\ \mathit{foldl'}\ \}\\
&\textbf{if}\ p\ x\ \textbf{then}\ a\ \textbf{else}\ b\ (c\ (tl\ x))\ \$!\ (a \oplus hd\ x)\\
=\quad &\{\ \text{definition of}\ \circ\ \}\\
&\textbf{if}\ p\ x\ \textbf{then}\ a\ \textbf{else}\ ((b \circ c)\ (tl\ x))\ \$!\ (a \oplus hd\ x)\\
=\quad &\{\ \text{definition of}\ \mathit{hylol'}\ \}\\
&\mathit{hylol'}\ (b \circ c)\ x\ a
\end{aligned}
$$

The functions *unfold′*, *foldl″* and *hylol′* are only locally defined above and so contain free variables, but are used in the above proof for simplicity.

Example: left factorial

Returning once again to our example, the factorial function can now be re-expressed using the fold-left version of the product function:

$$factl = productl \circ downFrom$$

Applying the left-hylomorphism theorem gives the following fused definition:

$$\begin{aligned} &factl = h\ 1 \\ &\quad \textbf{where}\ h\ a\ x = \textbf{if}\ x == 0\ \textbf{then}\ a\ \textbf{else}\ (h\ \$!\ (a * x))\ (x - 1) \end{aligned}$$

The resulting trace B in figure 6.2 shows that the multiplications now occur as soon as the list elements are generated. As a result the shape of the evaluation trace is different, because the evaluation now occurs in constant space, with only the additional space to hold the accumulator being required.

Strictness

The space performance of functions produced by the original hylomorphism theorem may in some cases still be constant. In particular, this occurs when the operator argument to the fold is non-strict in its second argument, in that the operator does not require the value of this argument in order to produce a result.

Example: prime

We can test if a natural number is prime by creating a list from two up to the given number, and checking that none of the elements are divisors:

$$\begin{aligned} &upto && :: Int \to [Int] \\ &upto\ n && = unfold\ (== n)\ id\ (+1)\ 2 \\ &nodivisors && :: Int \to [Int] \to Bool \end{aligned}$$

$$
\begin{aligned}
&\mathit{nodivisors}\ n = \mathit{foldr}\ (\lambda x\ b \to n\ `\mathit{mod}`\ x \neq 0 \wedge b)\ \mathit{True}\\
&\mathit{prime} \qquad\quad :: \mathit{Int} \to \mathit{Bool}\\
&\mathit{prime}\ n \qquad = (\mathit{nodivisors}\ n \circ \mathit{upto})\ n
\end{aligned}
$$

Applying the *hylor* theorem gives the following fused function:

$$
\begin{aligned}
&\mathit{prime}\ n = h\ 2\\
&\quad \textbf{where}\ h\ x = \textbf{if}\ x == n\ \textbf{then}\ \mathit{True}\ \textbf{else}\ n\ `\mathit{mod}`\ x \neq 0 \wedge h\ (x+1)
\end{aligned}
$$

In Haskell, conjunction ($\wedge$) is strict in its first argument, and non-strict in its second:

$$
\begin{aligned}
&\mathit{False} \wedge x = \mathit{False}\\
&\mathit{True} \wedge x\ = x
\end{aligned}
$$

Using this definition of $\wedge$, the evaluation trace for *prime* 9 is:

$$
\begin{aligned}
& \mathit{prime}\ 9\\
= \quad & h\ 2\\
= \quad & \mathit{True} \wedge h\ 3\\
= \quad & h\ 3\\
= \quad & \mathit{False} \wedge h\ 4\\
= \quad & \mathit{False}
\end{aligned}
$$

The resulting trace has constant space requirements, because $\wedge$ can be evaluated solely based on the value of its first argument. If the conjunction was implemented such that it was strict in both its arguments, then evaluation would occur as in the previous examples. Note that the fold-left version of the *prime* function still has constant space requirements, though the time requirements are worse if the number is not prime, because the fold-left always has a tail-recursive call, and hence it can never exploit the laziness of $\wedge$ if its first argument evaluates to *False*.

6.3 Formalisation

We now seek to formalise the space performance results of the previous section. The approach here, as described in the previous chapter, is to first transform the function

whose space performance we wish to measure into an abstract machine that makes explicit how evaluation proceeds. We then label the transitions of the machine with explicit space information, and reverse the transformation process to obtain a high-level function that measures the space behaviour of the original function. In the remainder of this section we show how this proceeds for the *hylor* function.

Abstract machines

Let us start with the definition of the *hylor* function:

$$
\begin{aligned}
&\mathit{hylor}\ p\ \mathit{hd}\ \mathit{tl}\ (\oplus)\ v = f \\
&\quad \textbf{where}\ f\ x = \textbf{if}\ p\ x\ \textbf{then}\ v\ \textbf{else}\ \mathit{hd}\ x \oplus f\ (\mathit{tl}\ x)
\end{aligned}
$$

The first step in the process of obtaining an abstract machine that implements this function is to make the control flow explicit, by transforming the function into continuation-passing style, giving the following result:

$$
\begin{aligned}
&\mathit{hylorCPS}\ p\ \mathit{hd}\ \mathit{tl}\ (\oplus)\ v\ x = h\ x\ \mathit{id} \\
&\quad \textbf{where}\ h\ x\ c = \textbf{if}\ p\ x\ \textbf{then}\ c\ v\ \textbf{else}\ h\ (\mathit{tl}\ x)\ (\lambda z \rightarrow c\ (\mathit{hd}\ x \oplus z))
\end{aligned}
$$

The next step is to replace the use of continuations by an explicit stack data structure by applying defunctionalization, which results in the following definition:

$$
\begin{aligned}
&\textbf{data}\ \mathit{Stack}\ a && = \mathit{TOP} \mid \mathit{PUSH}\ a\ (\mathit{Stack}\ a) \\
&\mathit{hyloMach}\ p\ \mathit{hd}\ \mathit{tl}\ (\oplus)\ v\ x && = h\ x\ \mathit{TOP} \\
&\quad \textbf{where}\ h\ x\ c && = \textbf{if}\ p\ x \\
&\qquad\qquad && \quad\ \textbf{then}\ \mathit{exec}\ c\ v \\
&\qquad\qquad && \quad\ \textbf{else}\ h\ (\mathit{tl}\ x)\ (\mathit{PUSH}\ (\mathit{hd}\ x \oplus)\ c) \\
&\qquad \mathit{exec}\ \mathit{TOP}\ v && = v \\
&\qquad \mathit{exec}\ (\mathit{PUSH}\ \mathit{yop}\ c)\ z && = \mathit{exec}\ c\ (\mathit{yop}\ z)
\end{aligned}
$$

We can now rewrite this function in the form of transition rules for an abstract machine with two states. The state (x, c) corresponds to evaluating an expression using the function call $h\ x\ c$, and $\langle c, v\rangle$ to executing a stack using $\mathit{exec}\ c\ v$:

$$
\begin{array}{lll}
(x, c) & \rightarrow & \textbf{if } p\ x \textbf{ then } \langle c, v\rangle \textbf{ else } (tl\ x, PUSH\ (hd\ x \oplus)\ c) \\
\langle TOP, v\rangle & \rightarrow & v \\
\langle PUSH\ yop\ c, z\rangle & \rightarrow & \langle c, yop\ z\rangle
\end{array}
$$

Finally, we also specify the evaluation order of the **else** branch within these rules, by introducing explicit let bindings with strict semantics:

$$
\begin{array}{lll}
(x, c) & \rightarrow & \textbf{if } p\ x \\
& & \quad \textbf{then } \langle c, v\rangle \\
& & \quad \textbf{else let } t = tl\ x \\
& & \qquad \textbf{in let } y = hd\ x \\
& & \qquad\quad \textbf{in let } yop = (\oplus)\ y \\
& & \qquad\qquad \textbf{in } (t, PUSH\ yop\ c) \\
\langle TOP, v\rangle & \rightarrow & v \\
\langle PUSH\ yop\ c, z\rangle & \rightarrow & \langle c, yop\ z\rangle
\end{array}
$$

Space costs

For the purposes of assigning space costs we use the notation x_s to denote the space requirements for evaluating an expression x. In the case when x is a piece of data, this will be a non-negative integer representing the size of that data, which we measure by simply counting constructors. For example, the cost of the stack data structure, which is given by the size of the stack, is defined recursively as follows:

$$
\begin{array}{ll}
TOP_s & = 1 \\
(PUSH\ x\ c)_s & = 1 + x_s + c_s
\end{array}
$$

In the case of a function f with a single argument, the cost f_s will be a function that takes this argument along with a memory manager, and returns a modified memory manager that reflects the cost of this application. For example, the cost of applying the tail function on lists can be expressed as follows:

$$tail_s\ (x : xs) = free\ (1 + x_s)$$

The argument to the tail function is a list $x : xs$, and the result is a list xs. Therefore applying the *tail* function results in freeing the memory occupied by the list con-

structor :, which is one unit, and the element x, which is of size x_s. Functions with multiple arguments can be treated in the same way by exploiting currying, resulting in a function of n arguments having n unary cost functions.

Transition costs

To add space information to the derived abstract machine we use the same method as in the previous chapter. However the cost function, that was defined previously by first freeing any space not required and then allocating space for any new structures, needs to be extended because the machine in this instance has two special cases to consider, namely when transitions introduce a **let** or an **if**.

For transitions of the form $x \ \rightarrow \ \textbf{let}\ y = f\ x'\ \textbf{in}\ z$, we first allocate the space for the argument x', then the space requirements for the application $f\ x'$, and finally apply the *cost* function to the sizes of the left and right-hand expressions:

$$\mathit{cost}\ (x \ \rightarrow \ \textbf{let}\ y = f\ x'\ \textbf{in}\ z) = \mathit{cost}\ (x_s + y_s)\ z_s \circ f_s\ x' \circ \mathit{alloc}\ {x'}_s$$

Similarly in the **if** case, the space cost of performing the transition $x \ \rightarrow \ \textbf{if}\ p\ x'\ \textbf{then}\ y\ \textbf{else}\ z$ first allocates the space for x', then applies the cost of the application $p\ x'$. If the result is *True* then the cost function is applied with the size of the left-hand-side $x_s + \mathit{True}_s$ and right-hand-side y_s, and if it is *False* then the size of the left-hand-side is $x_s + \mathit{False}_s$ and right-hand-side z_s:

$$\begin{aligned}&\mathit{cost}\ (x \ \rightarrow \ \textbf{if}\ p\ x'\ \textbf{then}\ y) = \\ &(\textbf{if}\ p\ x'\ \textbf{then}\ \mathit{cost}\ (x_s + \mathit{True}_s)\ y_s\ \textbf{else}\ \mathit{cost}\ (x_s + \mathit{False}_s)\ z_s) \circ p_s\ x' \circ \mathit{alloc}\ {x'}_s\end{aligned}$$

In the new machine each argument is paired with its space cost. For example x is replaced by (x, x_s). The resulting machine, which has also been simplified by inlining the definition of *cost* and applying the properties in section 5.3, is given below:

$$\begin{aligned}&\mathit{spaceMach}\ (p, p_s)\ (hd, hd_s)\ (tl, tl_s)\ ((\oplus), op_{s1}, op_{s2})\ v\ (x, x_s)\ m = \\ &\quad h\ x\ (\mathit{alloc}\ 1\ m)\ \mathit{TOP} \\ &\qquad \textbf{where}\ h\ x\ m\ c = \\ &\qquad\quad \textbf{if}\ p\ x\end{aligned}$$

$$
\begin{array}{l}
\quad \textbf{then } \mathit{exec}\ c\ v\ ((\mathit{free}\ (x_s + 1) \circ p_s\ x \circ \mathit{alloc}\ (x_s))\ m) \\
\quad \textbf{else let } t = \mathit{tl}\ x \\
\qquad\qquad\quad y = \mathit{hd}\ x \\
\qquad\qquad\quad \mathit{yop} = (\oplus)\ y \\
\qquad\quad \textbf{in } h\ t\ ((\mathit{alloc}\ 1 \circ \mathit{free}\ x_s \circ \mathit{op}_{s1}\ y \circ \mathit{hd}_s\ x \circ \mathit{alloc}\ x_s \circ \mathit{tl}_s\ x \circ \\
\qquad\qquad\quad \mathit{alloc}\ x_s \circ \mathit{free}\ 1 \circ p_s\ x \circ \mathit{alloc}\ x_s)\ m)\ (\mathit{PUSH}\ \mathit{yop}\ c) \\
\mathit{exec}\ \mathit{TOP}\ v\ m \qquad\qquad\quad = \mathit{free}\ 1\ m \\
\mathit{exec}\ (\mathit{PUSH}\ \mathit{yop}\ c)\ z\ m = \mathit{exec}\ c\ (\mathit{yop}\ z)\ ((\mathit{free}\ 1 \circ \mathit{op}_{s2}\ z)\ m)
\end{array}
$$

The next step is to perform the same program transformations, but in the reverse order, to produce a high-level function that measures the space from the abstract machine. After refunctionalizing the continuation and transforming into direct style, the following accumulator version is produced:

$$
\begin{array}{l}
\mathit{spacer}\ (p, p_s)\ (\mathit{hd}, \mathit{hd}_s)\ (\mathit{tl}, \mathit{tl}_s)\ ((\oplus), \mathit{op}_{s1}, \mathit{op}_{s2})\ v\ (x, x_s) = \\
\quad \mathit{free}\ 1 \circ h\ x \circ \mathit{alloc}\ 1 \\
\qquad \textbf{where } h\ x = \\
\qquad\quad \textbf{if } p\ x \\
\qquad\quad \textbf{then } \mathit{free}\ (x_s + 1) \circ p_s\ x \circ \mathit{alloc}\ x_s \\
\qquad\quad \textbf{else let } t = \mathit{tl}\ x \\
\qquad\qquad\quad z = \mathit{hylor}\ p\ \mathit{hd}\ \mathit{tl}\ (\oplus)\ v\ t \\
\qquad\qquad \textbf{in } \mathit{free}\ 1 \circ \mathit{op}_{s2}\ z \circ h\ t \circ \mathit{alloc}\ 1 \circ \mathit{free}\ x_s \circ \mathit{op}_{s1}\ (\mathit{hd}\ x) \circ \mathit{hd}_s\ x \circ \\
\qquad\qquad\quad \mathit{alloc}\ x_s \circ \mathit{tl}_s\ x \circ \mathit{alloc}\ x_s \circ \mathit{free}\ 1 \circ p_s\ x \circ \mathit{alloc}\ x_s
\end{array}
$$

In the next section, we will use this derived function to verify the space requirements of our two versions of the factorial function.

Example: factorial

We can analyse the space performance of the factorial function by first producing space requirements functions for the primitive functions that are used in its definition, namely $(== 0)$, $*$ and *pred*. This is performed simply by taking the difference in size between the argument and result. For example, if we define the size of an integer to

be one unit of space, then the $*$ operator will free one unit of space, because it takes two integers as arguments and returns a single integer as the result.

Applying the *spacer* function and inlining the primitive space functions gives the following space requirements for the factorial function:

$$\begin{aligned} &\mathit{spaceFact}\ x \quad = \mathit{free}\ 1 \circ f\ x \circ \mathit{alloc}\ 1 \\ &\quad \textbf{where}\ f\ x = \textbf{if}\ (x == 0)\ \textbf{then}\ \mathit{free}\ 2 \circ \mathit{alloc}\ 1\ \textbf{else}\ \mathit{free}\ 2 \circ f\ (x-1) \circ \mathit{alloc}\ 2 \end{aligned}$$

The resulting function shows how, for each recursive call to f, two units need to be allocated before the call, which are then released afterwards.

We can prove that the space requirements for the factorial function, *spaceFact* x, is linear by showing that it executes in $2*x$ units of additional space. This is achieved by proving the following equation, which states that if we free $2 * x$ units of space and then execute the *spaceFact* function, then the allocated amount of memory will be unchanged. This means there was no need to request more memory, because the pool of $2 * x$ unit of free memory was sufficient for evaluation:

$$\mathit{allocated} \circ \mathit{spaceFact}\ x \circ \mathit{free}\ (2 * x) = \mathit{allocated}$$

Proof: by induction over the natural number x.

Case : 0

$$\begin{aligned} & \mathit{allocated} \circ \mathit{spaceFact}\ 1 \circ \mathit{free}\ 2 \\ = & \quad \{\ \text{definition of } \mathit{spaceFact}\ \} \\ & \mathit{allocated} \circ \mathit{free}\ 1 \circ f\ 1 \circ \mathit{alloc}\ 1 \circ \mathit{free}\ 2 \\ = & \quad \{\ \text{definition of } f\ \} \\ & \mathit{allocated} \circ \mathit{free}\ 1 \circ \mathit{free}\ 2 \circ \mathit{alloc}\ 1 \circ \mathit{alloc}\ 1 \circ \mathit{free}\ 2 \\ = & \quad \{\ \text{Property 3: } \mathit{alloc}\ n \circ \mathit{free}\ n = \mathit{id}\ \} \\ & \mathit{allocated} \circ \mathit{free}\ 1 \circ \mathit{free}\ 2 \\ = & \quad \{\ \text{Property 1: } \mathit{free}\ n \circ \mathit{free}\ m = \mathit{free}\ (m+n)\ \} \\ & \mathit{allocated} \circ \mathit{free}\ 3 \\ = & \quad \{\ \text{Property 4: } \mathit{allocated} \circ \mathit{free}\ n = \mathit{allocated}\ \} \\ & \mathit{allocated} \end{aligned}$$

Case : $x + 1$

$$
\begin{aligned}
& \mathit{allocated} \circ \mathit{spaceFact}\ (x+1) \circ \mathit{free}\ (2 + 2 * x) \\
= & \quad \{\ \text{definition of } \mathit{spaceFact}\ \} \\
& \mathit{allocated} \circ \mathit{free}\ 1 \circ f\ (x+1) \circ \mathit{alloc}\ 1 \circ \mathit{free}\ (2 + 2 * x) \\
= & \quad \{\ \text{definition of } f\ \} \\
& \mathit{allocated} \circ \mathit{free}\ 1 \circ \mathit{free}\ 2 \circ f\ x \circ \mathit{alloc}\ 2 \circ \mathit{alloc}\ 1 \circ \mathit{free}\ (2 + 2 * x) \\
= & \quad \{\ \text{Property 3: } \mathit{alloc}\ n \circ \mathit{free}\ n = \mathit{id}\ \} \\
& \mathit{allocated} \circ \mathit{free}\ 1 \circ \mathit{free}\ 2 \circ f\ x \circ \mathit{alloc}\ 1 \circ \mathit{free}\ 2 * x \\
= & \quad \{\ \text{definition of } \mathit{spaceFact}\ \} \\
& \mathit{allocated} \circ \mathit{free}\ 2 \circ \mathit{spaceFact} \circ \mathit{free}\ 2 * x \\
= & \quad \{\ \text{induction hypothesis}\ \} \\
& \mathit{allocated}
\end{aligned}
$$

We can perform a similar calculation for the left factorial version, using the left-hylo function. The function *hylol* is already an abstract machine, since it is tail-recursive and first order, so we can directly annotate the function with space rules in the same way, to produce the following function:

$$
\begin{aligned}
& \mathit{spacel}\ (p, p_s)\ (hd, hd_s)\ (tl, tl_s)\ ((\oplus), op_{s1}, op_{s2})\ v\ (x, x_s) = h\ v\ x \\
& \quad \textbf{where}\ h\ a\ x = \textbf{if}\ p\ x \\
& \qquad\qquad \textbf{then}\ \mathit{free}\ (x_s + 1) \circ p_s\ x \circ \mathit{alloc}\ x_s \\
& \qquad\qquad \textbf{else}\ h\ (a \oplus hd\ x)\ (tl\ x) \circ tl_s\ x \circ op_{s2}\ a \circ op_{s1}\ (hd\ x) \circ \\
& \qquad\qquad\qquad\qquad hd_s\ x \circ \mathit{alloc}\ x_s \circ \mathit{free}\ 1 \circ p_s\ x \circ \mathit{alloc}\ x_s
\end{aligned}
$$

Applying the space requirements function and inlining the primitive functions gives the following result for the left factorial function:

$$
\begin{aligned}
& \mathit{spaceFactl} \quad = h \\
& \quad \textbf{where}\ h\ x = \textbf{if}\ x == 0\ \textbf{then}\ \mathit{free}\ 2 \circ \mathit{alloc}\ 1\ \textbf{else}\ h\ (x-1) \circ \mathit{free}\ 2 \circ \mathit{alloc}\ 2
\end{aligned}
$$

In this function, for each recursive call two units of space are allocated but then freed before the call, so there is no overhead. We can prove that the additional space for evaluating *spaceFactl* x is just two units, using the following equation:

$$allocated \circ spaceFactl\ x \circ free\ 2 = allocated$$

Proof: by induction over the natural number x.

Case : 0

$$\begin{aligned}
& allocated \circ spaceFactl\ 1 \circ free\ 2 \\
= & \quad \{\ \text{definition of } spaceFactl\ \} \\
& allocated \circ free\ 2 \circ alloc\ 1 \circ free\ 2 \\
= & \quad \{\ \text{Property 3: } alloc\ n \circ free\ n = id\ \} \\
& allocated \circ free\ 2 \circ free\ 1 \\
= & \quad \{\ \text{Property 1: } free\ n \circ free\ m = free\ (m+n)\ \} \\
& allocated \circ free\ 3 \\
= & \quad \{\ \text{Property 4: } allocated \circ free\ n = allocated\ \} \\
& allocated
\end{aligned}$$

Case : $x + 1$

$$\begin{aligned}
& allocated \circ spaceFactl\ (x+1) \circ free\ 2 \\
= & \quad \{\ \text{definition of } spaceFactl\ \} \\
& allocated \circ spaceFactl\ x \circ free\ 2 \circ alloc\ 2 \circ free\ 2 \\
= & \quad \{\ \text{Property 3: } alloc\ n \circ free\ n = id\ \} \\
& allocated \circ spaceFactl\ x \circ free\ 2 \\
= & \quad \{\ \text{induction hypothesis}\ \} \\
& allocated
\end{aligned}$$

Example: converting to binary

A natural number n can be converted into binary form as a list of zeros and ones by repeatedly dividing the number by two and taking the remainder. This process can naturally be expressed using *unfold*, as follows:

$$\begin{aligned}
& toBinSigRight :: Int \to [Int] \\
& toBinSigRight = unfold\ (== 0)\ (`mod`\ 2)\ (`div`\ 2)
\end{aligned}$$

The result of *toBinSigRight* is a list where the most significant bit is on the right, which can then be reversed to give the standard representation. The process of reversing a list can itself be written as a fold-right:

$$reverse' = foldr\ (\lambda x\ xs \rightarrow xs +\!\!+ [x])\ [\,]$$

The composition of these two definitions yields a function that converts a natural number into binary form, with the most significant bit on the left:

$$toBin = reverse' \circ toBinSigRight$$

For example, $toBin\ 10 = [1, 0, 1, 0]$. Applying the *hylor* theorem gives the following fused definition that eliminates the intermediate list:

```
toBin :: Int → [Int]
toBin = h
  where h x = if x == 0 then [] else h (x `div` 2) ++ [x `mod` 2]
```

It is not possible for evaluation of *toBin* to occur in constant space, because the result is a list of non-constant size, but we can look at the additional space required to produce the result. Applying the space function yields the following result:

```
spaceToBin n = free 1 ∘ h n ∘ alloc 1
  where h n = if n == 0 then free 2 ∘ alloc 1 else free 2 ∘ h (n `div` 2) ∘ alloc 4
```

From this, we can prove the minimum additional space required for evaluation is $6 + 4 * \log_2 n$. The size of the result is a list sized $1 + 2 * \log_2 n$ (one unit for each bit and list-constructor plus an additional unit to hold the empty list). Excluding this space, the evaluation requires an additional $5 + 2 * \log_2 n$ units. We can compare this to the accumulating version by re-expressing the reverse function using fold-left.

Another fold duality theorem [4] states that a fold-right may be rewritten as a fold-left if their operators associate with each other, and the empty list value is the right and left unit for the fold-right and fold-left operator respectively:

$$\begin{array}{l} x \oplus (y \otimes z) = (x \oplus y) \otimes z \\ x \oplus v = v \otimes x \end{array} \quad \Rightarrow \quad foldr\ (\oplus)\ v\ xs = foldl\ (\otimes)\ v\ xs$$

Because $(z : y) +\!\!+ [x] = z : (y +\!\!+ [x])$ and $[\,] +\!\!+ [x] = x : [\,]$, we can apply this theorem to re-express the *reverse* function using *foldl*:

$$reversel = foldl\ (flip\ (:))\ [\,]$$

In turn, if we redefine the *toBin* function using *reversel*, and apply the *hylol* rule, the result is the following fused definition:

$$\begin{aligned} &toBinl = g\ [\,] \\ &\quad \textbf{where}\ g\ a\ x = \textbf{if}\ x == 0\ \textbf{then}\ a\ \textbf{else}\ (g\ \$!\ (x\ `mod`\ 2 : a))\ (x\ `div`\ 2) \end{aligned}$$

In the same manner as previously, the left-hylo space function can then be applied to produce the following space requirements function:

$$\begin{aligned} &spaceToBinl = h \\ &\quad \textbf{where}\ h\ x = \textbf{if}\ x == 0\ \textbf{then}\ free\ 3 \circ alloc\ 2\ \textbf{else}\ h\ (x\ `div`\ 2) \circ alloc\ 2 \end{aligned}$$

This function gives the space requirements $4 + 2 * \log_2 n$. However, excluding the space that the result is occupying, it only requires a constant 3 units for evaluation.

The results of the two space requirements functions show that the right-hylo version requires additional space proportional to the size of the result, whereas the left version only requires a constant amount of additional space.

6.4 Summary

The aim of applying fusion theorems, such as the hylomorphism theorem, is to eliminate intermediate data structures. However, we have shown that this is only achieved if the generating function produces elements in the *same order* as they are consumed. The examples given illustrate this impedance mismatch and show how an accumulator version, defined using fold-left, can provide a solution. Using the accumulator the list elements can be evaluated as they are generated, rather than waiting until the whole list structure has been created, resulting in improved space performance.

The space results were first observed informally by looking at evaluation traces, but more accurate space measures can be obtained by using program transformation

techniques to derive the underlying abstract machine. At this level we were able to measure the size of data structures that were not visible at the original function level. The resulting machine was then instrumented with space information, and the transformations reversed to produce a space requirements function. This function was then used to prove the space requirements of both the original and improved versions of the factorial function.

Applying this technique to more general structures than lists may be problematic, because fold-left cannot be readily generalised as fold-right can. There are more restrictive functions that can be generalised, such as *crush* [60], where the structure is first flattened to a list and then folded. The same idea of using an accumulating fold could then be applied to improve the space usage. How to extend this approach to other structures would be an interesting topic for further work.

Chapter 7

Conclusion

In this final chapter we briefly summarise the contents of the thesis, together with the primary contributions of the work. We then consider a number of directions for further work, in particular how the techniques may be extended to lazy evaluation, and how the derivation process may be automated.

7.1 Summary

Chapter 1 gave an introduction to the area of space and time analysis, starting with the standard approach for imperative languages, followed by a discussion on the additional complexities that arise for functional languages. Some simple examples were used to illustrate these complexities, and the chapter concluded with a survey of related work in the area of reasoning about the performance of functional programs.

Chapter 2 demonstrated a conventional approach to space and time analysis, using an instrumented operational semantics. Two different forms of operational semantics were considered, a small-step semantics to capture the notion of a single step of evaluation, and an equivalent big-step semantics that is more suitable for the purposes of reasoning about the time performance of programs. However it is not clear how the time information from such semantics relates to the actual process of executing programs. Moreover, these semantics did not provide access to any of the hidden data structures required for evaluation, and are therefore not well suited for

considering space performance. The combination of these two limitations provided the motivation to instead consider lower-level implementations. The approach we take in this thesis is to consider abstract machines, which provide a good compromise between a high-level semantics and a low-level implementation.

In chapter 3 we introduced the concept of abstract machines and starting with a denotational semantics for a language, showed how an abstract machine that implements the same semantics can be calculated. This machine was then instrumented with time information in chapter 4, corresponding to the number of transitions required, and a high-level function that measures the time requirements calculated in the same manner. The resulting function was then used to formally justify the time requirements of the functions considered in the first chapter.

Building on our earlier approach to time analysis, chapter 5 showed how an abstract machine can be extended to measure space requirements, by considering the additional evaluation structures that are revealed at the machine level. The abstract machine derived in chapter 3 was instrumented using a memory manager to keep track of not only the amount of allocations required, but also the space deallocated, in order that this can be re-used later on. These space analysis techniques were applied to an extended example in chapter 6, namely that of obtaining a more space efficient version of the standard hylomorphism theorem.

In conclusion, the primary contribution of this thesis is a new approach to producing space and time requirements for functional programs, that provably corresponds to information at the level of an underlying abstract machine. Although using abstract machines to obtain space and time information has been considered in previous work, we are the first to show how to calculate this information directly from an evaluator. We have also shown how to instrument the intermediate abstract machines with time and space information, and how the resulting machines can be re-transformed back to a high-level function capturing time and space information. Finally, we demonstrated how our space analysis techniques could be applied to the hylomorphism operator on lists. In particular, we presented a new hylomorphism theorem, expressed using fold-left rather than fold-right, and verified using our abstract machine approach that this

has better space performance than the standard version.

7.2 Further work

There are a number of interesting directions for further work. First of all, it would be natural to consider extending the language in this thesis with additional features, such as other data structures and recursion operators, to facilitate considering more sophisticated programming examples, for example algorithms on tree structures. Given the foundational work laid out in this thesis, we would anticipate that such a generalisation would be relatively straightforward from a development point of view.

In the remainder of this section, we expand on two other directions for further work that are more interesting from a research point of view: the use of lazy evaluation, and automation of the calculation process.

7.2.1 Lazy evaluation

As discussed in section 1.3, lazy evaluation is the combination of call-by-name evaluation with sharing of common sub-expressions. In order to express sharing in a high-level semantics, an additional data structure is required to hold the common sub-expressions, to which the expression being evaluated can point to.

Lazy semantics

The big step semantics for lazy evaluation presented by Launchbury [61] introduces a heap to a substitution-based call-by-name semantics for an extended lambda-calculus. The heap is a partial function from variables to expressions, which means that variables in the expression under evaluation can be considered as pointers into the heap.

Sharing is captured in the language using let-expressions, such that an expression **let** $x = y$ **in** e is evaluated by associating x on the heap to the expression y, and evaluating e with this extended heap. If the variable x is encountered when evaluating

e, the heap is dereferenced for x and then y is evaluated. Once y has been evaluated, x is updated on the heap to point to its value, so that subsequent lookups of x will no longer need to evaluate y again.

In order to be able to reclaim space, and therefore provide a more realistic model, garbage collection of the heap needs to be considered. The semantics can be extended with extra rules that may be applied at any time to discard any unnecessary heap elements. One possible garbage collection rule could be implemented by maintaining a list of variables still currently active. Periodically the heap could then be cleaned to remove any bindings for variables that are not reachable from the list.

Lazy machines

The motivation of Launchbury's work was to provide a "common semantic base" that was independent of any particular abstract machine implementation. Sestoft [62] has applied this big-step semantics to various abstract machine implementations. The approach Sestoft takes is to invent rather than calculate the abstract machines, and then prove they are correct with respect to the semantics by a combination of the notion of balanced computation, and induction on the length of the computation.

Sestoft's first machine, the "mark 1", adds a control structure to the original evaluation semantics, to result in a lazy abstract machine that uses substitution. This machine has formed the basis for several works on time and space analysis [30, 20, 33]. The subsequent machines utilise an environment, where the environment is a mapping between variables and pointers onto the heap, and have also considered the use of de Bruijn indices instead of explicit variable names.

Sestoft also discusses the space requirements of these machine implementations, in particular the leak produced by maintaining an environment that binds more variables than required. In order to address this problem, environment trimming is considered, which may be implemented by annotating every let-expression with its set of free variables. In this manner, the environment could then be trimmed by restricting the domain of the environment to this set of variables.

Deriving lazy machines

The work of Danvy *et al* in deriving machines from evaluators has been applied to consider lazy evaluation [46], starting with a call-by-name evaluator that is extended by threading an updateable heap. An abstract machine is derived in precisely the same manner as in Danvy's previous work. In particular, the evaluator is CPS-transformed, and the continuation is then defunctionalized to produce an abstract machine. However, the resulting machine does not aim to provide an efficient space model, in that it does not implement trimming the environment, or removing excess structures from the heap in order that space can be re-used.

Adapting our approach

In this section we outline how the approach developed in this thesis could be extended to consider lazy evaluation. As a first step we would need to extend our language with a **let** construct, to allow sharing to be made explicit in the language. The next step would be to redefine the evaluator to implement call-by-name evaluation, and introduce the notion of a heap to allow sharing to be made explicit.

To re-use space on the heap once it is no longer required, we would need to address the issue of garbage collection. In this thesis, the use of a substitution-based model for evaluation had the advantage of making it possible to recover space as part of the evaluation process itself, without the need for a separate garbage collection process. However, moving to lazy evaluation introduces an additional structure in the form of the heap, which would necessitate the introduction of a garbage collector to identify parts of the heap that are no longer in scope and can be reclaimed.

If we adopt a substitution-based evaluator for our lazy language, then variables in the expression being evaluated will point directly into the heap. In this setting, we would need to maintain a reference count for each heap location, in order that garbage collection can determine when a structure is no longer required, and reclaim this space. Alternatively, if the evaluator is environment-based, then the environment is a mapping from variables to heap locations, and hence we also need to consider the

notion of environment-trimming to reclaim any space in the environment that is no longer required.

Given a suitable lazy evaluator, we could then apply the techniques developed in this thesis to calculate an abstract machine, add the necessary instrumentation, and finally reverse the calculation process to obtain an instrumented evaluator, which can then be used as the basis for reasoning about space and time performance. However, the result would be considerably more complicated than for call-by-value evaluation, due to the demand driven nature of lazy evaluation and the additional complexities of garbage collection, and hence more difficult to apply in practice. Nonetheless, exploring how our techniques could be extended to lazy languages is an interesting and important topic for further work.

7.2.2 Automation

The calculations presented in this thesis were produced by hand, which is a time-consuming and error-prone activity. To make this method of analysis more practical it would be useful to consider some form of mechanical support. There are two approaches that may be taken to achieve this: mechanical assistance for the calculation process, or complete automation of one or more of the component transformations.

Mechanical assistance

A number of reasoning assistants have been produced for Haskell, of which the most readily applicable to the work in this thesis is the Haskell Equational Reasoning Assistant (HERA) [63]. HERA is a graphical tool that allows the user to select a fragment of code, and then apply a selected transformation rule to this fragment. The system provides a large number of built-in transformation rules, which implement familiar transformations such as applying definitions, simplifying expressions, and case analysis. It also keeps track of properties that remain to be proved, any assumptions which have yet to be discharged, and is extensible in that the user can specify their own transformation rules. Using a system such as HERA could alleviate much of

the book-keeping tasks that form part of program calculation, and also ensure that mistakes are not made when applying substitutions. Investigating how our techniques could be applied within the HERA system is an interesting topic for further work.

A more sophisticated approach to mechanising the calculation process would be to utilise a modern theorem proving system such as Agda [64], in which programs and their proofs inhabit a single uniform framework, through the use of dependent types. Recent work has established the feasibility of performing program calculation within Agda, by means of a special purpose equational reasoning library developed by Norell [65] and Danielsson [64]. This library provides a range of combinators that allow the definition of a function to be constructed alongside a proof of its correctness. During this process the sophisticated type system underlying Agda assists in determining the appropriate next step, such as if an inductive assumption may be applied. Using this library, Mu, Ko and Jansson [66] have shown how insertion sort may be derived from a high-level specification of its behaviour.

This Agda library has recently been applied by Danielsson [67] to derive an abstract machine for the simple language of integers and addition as presented in section 3.1. The process proceeds by starting with a specification that expresses the requirements of the function we seek to derive. For example, the goal of applying the CPS transformation to the original evaluator $eval :: Expr \to Value$ is to find a new function $eval' :: Cont \to Expr \to Value$ such that $c\ (eval\ e) = eval'\ c\ e$. To express this goal in Agda we combine the two separate concepts of type and specification into a single dependent type. This type expresses that the new function $eval'$ takes a continuation and an expression, and returns a value that is equal to the result of evaluating the expression and then applying the continuation. More formally, we seek to construct a function $eval'$ with the following dependent type:

$$eval' :: Cont \to Expr \to exists\ v.v = c\ (eval\ e)$$

The use of an existential quantifier combines the type with the specification, and allows the definition of the function to be built alongside the proof of its correctness. Each step in this process must be guided by the programmer, but proceeds in the same manner as in the thesis, with only minor syntactic differences. While the use of

Agda to support program calculation is still in the early stages of development, this seems to be a promising approach for mechanising the work in this thesis.

One particularly interesting aspect of using Agda to mechanise the derivation of abstract machines is the role that totality plays. In particular, the fact that Agda is a total language means that all definitions must be verifiably terminating, which in practice is achieved by the restriction to structurally recursive definitions. However, even for the simple case of integers and addition, the derived abstract machine is not structurally recursive, and hence the type checker can not verify that it will terminate. In such circumstances, a separate proof of termination is required. In the case of Danielsson's derivation, such a proof amounts to augmenting the machine with an additional numeric value, which corresponds precisely to our step-counting results for this language. The connection between totality of an abstract machine and our work on step-counting is an interesting topic for further investigation.

Complete automation

To completely automate the derivation process, existing techniques for automating each of the individual program transformation may be employed, and possibly fused to short-cut the derivation process. For the first phase of our derivation process, the CPS transformation has been implemented by Plotkin [68] for the call-by-value lambda calculus, and proved to be correct by Dargaye and Leroy [69]. The implementation consists of a transformation rule for each part of the language syntax, which are applied recursively to achieve the desired transformation into CPS.

For the second phase of our derivation process, defunctionalization has been implemented by Reynolds [47], and proved to be correct by Banerjee, Heintze and Riecke [70]. In contrast to the CPS transformation, which is a standard recursive algorithm, the defunctionalization algorithm proceeds by recursion on the typing derivation of the original program, rather than its syntactic structure.

We anticipate that it would be relatively straightforward to implement an algorithm for instrumenting the resulting abstract machine with time or space information

in the manner described in this thesis. Implementing the reverse process of refunctionalization and transformation into direct style would require further work, but has the potential to lead to a fully automated implementation of our approach to analysing the space and time performance of functional programs.

References

[1] Roland Backhouse. *Program Construction: Calculating Implementations from Specifications*. John Wiley, 2003.

[2] Simon Peyton Jones. Haskell 98 language and libraries: The revised report. Technical report, Cambridge University Press, 2003.

[3] Clifford A. Shaffer. *A practical introduction to data structures and algorithms analysis, Java edition*. Prentice-Hall, 1998.

[4] Richard Bird. *Introduction to Functional Programming using Haskell*. Series in Computer Science. Prentice-Hall, second edition, 1998.

[5] Ben Wegbreit. Mechanical program analysis. *Commun. ACM*, 18(9):528–539, 1975.

[6] Daniel Le Metayer. Mechanical analysis of program complexity. In *Proceedings of the ACM SIGPLAN 85 symposium on Language issues in programming environments*, pages 69–73, New York, NY, USA, 1985. ACM Press.

[7] Mads Rosendahl. Automatic complexity analysis. In *FPCA '89: Proceedings of the fourth international conference on Functional programming languages and computer architecture*, pages 144–156, New York, NY, USA, 1989. ACM Press.

[8] Yanhong A. Liu and Gustavo G'omez. Automatic accurate time-bound analysis for high-level languages. *Lecture Notes in Computer Science*, 1474:31–50, 1998.

[9] Gustavo G'omez and Yanhong A. Liu. Automatic time-bound analysis for a higher-order language, 1999.

[10] Ralph Benzinger. *Automated computational complexity analysis*. PhD thesis, Cornell University, 2001. Chair-Robert L. Constable.

[11] Ralph Benzinger. Automated higher-order complexity analysis. *Theor. Comput. Sci.*, 318(1-2):79–103, 2004.

[12] Philip Wadler. Strictness analysis aids time analysis. In *POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 119–132, New York, NY, USA, 1988. ACM Press.

[13] David Sands. Complexity analysis for a higher order language. Technical Report DOC 88/14, Imperial College, October 1988.

[14] David Sands. Complexity analysis for a lazy higher-order language. In *Proceedings of the third European symposium on programming on ESOP '90*, pages 361–376, New York, NY, USA, 1990. Springer-Verlag New York.

[15] David Sands. A naïve time analysis and its theory of cost equivalence. *Journal of Logic and Computation*, 5(4), 1995.

[16] Bror Bjerner and S. Holmström. A composition approach to time analysis of first order lazy functional programs. In *FPCA '89: Proceedings of the fourth international conference on Functional programming languages and computer architecture*, pages 157–165, New York, NY, USA, 1989. ACM.

[17] Brian Reistad and David K. Gifford. Static dependent costs for estimating execution time. *SIGPLAN Lisp Pointers*, VII(3):65–78, 1994.

[18] A.J.R. Portillo, K. Hammond, H.W. Loidl, and P. Vasconcelos. Cost analysis using automatic size and time inference. *Proc. IFL 2002–Implementation of Functional Languages, Madrid, Spain*, 2002.

[19] Kevin Hammond and Pedro Vasconcelos. Inferring costs for recursive, polymorphic and higher-order functional programs, 2003. Submitted to 2004 ACM Symposium on Principles of Programming Languages – POPL 2004.

[20] Andrew Moran and David Sands. Improvement in a lazy context: An operational theory for call-by-need. In *Conference Record of POPL 99: The 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, Texas*, pages 43–56, New York, NY, 1999.

[21] Nils Anders Danielsson. Lightweight semiformal time complexity analysis for purely functional data structures. *SIGPLAN Not.*, 43(1):133–144, 2008.

[22] Greg Morrisett, Matthias Felleisen, and Robert Harper. Abstract models of memory management. In *FPCA '95: Proceedings of the seventh international conference on Functional programming languages and computer architecture*, pages 66–77, New York, NY, USA, 1995. ACM.

[23] Greg Morrisett and Robert Harper. Semantics of memory management for polymorphic languages. In A. Gordon and A. Pitts, editors, *Higer Order Operational Techniques in Semantics*. Newton Institute, Cambridge University Press, 1997.

[24] John Hughes and Lars Pareto. Recursion and dynamic data-structures in bounded space: Towards embedded ML programming. In *International Conference on Functional Programming*, pages 70–81, 1999.

[25] Adam Bakewell and Colin Runciman. A space semantics for core Haskell. In *2000 ACM SIGPLAN Haskell Workshop, Montreal, Canada*, volume 41 of *Electronic Notes in Theoretical Computer Science*, 2001.

[26] Yasuhiko Minamide. Space-profiling semantics of the call-by-value lambda calculus and the CPS transformation. *Electronic Notes Theoretical Computer Science*, 26, 1999.

[27] Guy E. Blelloch and John Greiner. A provable time and space efficient implementation of NESL. In *ICFP '96: Proceedings of the first ACM SIGPLAN*

international conference on Functional programming, pages 213–225, New York, NY, USA, 1996. ACM.

[28] Martin Hofmann and Stefan Jost. Static prediction of heap space usage for first-order functional programs, 2003.

[29] Olha Shkaravska. Amortized heap-space analysis for first-order functional programs. *In M. van Eekelen, ed., Proc. of 6th Symp. on Trends in Functional Programming, TFP '05*, pages 281–296, Sept 2005.

[30] Jrgen Gustavsson and David Sands. A foundation for space-safe transformations of call-by-need programs. In *Proceedings of Workshop on Higher Order Operational Techniques in Semantics*, number volume 26 of Electronic Notes in Theoretical Computer Science, September 1999.

[31] Jrgen Gustavsson and David Sands. Possibilities and limitations of call-by-need space improvement. In *ICFP '01: Proceedings of the sixth ACM SIGPLAN international conference on Functional programming*, pages 265–276. ACM Press, 2001.

[32] Martin Hofmann. A type system for bounded space and functional in-place update. *Nordic Journal of Computing*, 7(4):258–289, Autumn 2000.

[33] Patrick M. Sansom and Simon Peyton Jones. Formally based profiling for higher-order functional languages. *ACM Trans. Program. Lang. Syst.*, 19(2):334–385, 1997.

[34] Colin Runciman and David Wakeling. Heap profiling of lazy functional programs. *Journal of Functional Programming*, 3(2):217–245, 1993.

[35] Alonzo Church. *The Calculi of Lambda Conversion. (AM-6) (Annals of Mathematics Studies)*. Princeton University Press, Princeton, NJ, USA, 1985.

[36] David A. Schmidt. *Denotational semantics: a methodology for language development*. William C. Brown Publishers, Dubuque, IA, USA, 1986.

[37] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, University of Aarhus, 1981.

[38] Carl A. Gunter. *Semantics of Programming Languages: Structures and Techniques*. Foundations of Computing. MIT Press, 1992.

[39] Matthias Felleisen. *The Calculi of λ-ν-CS Conversion: A Syntactic Theory of Control And State in Imperative Higher-Order Programming Languages*. PhD thesis, August 1987.

[40] Graham Hutton. A tutorial on the universality and expressiveness of fold. *Journal of Functional Programming*, 9(4):355–372, July 1999.

[41] Simon Peyton Jones. Implementing lazy functional languages on stock hardware: The spineless tagless g-machine. *Journal of Functional Programming*, 2(2):127–202, 1992.

[42] Graham Hutton and Joel Wright. Calculating an exceptional machine. In Hans-Wolfgang Loidl, editor, *Trends in Functional Programming volume 5*. Intellect, February 2006. Selected papers from the Fifth Symposium on Trends in Functional Programming, Munich, November 2004.

[43] Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. A functional correspondence between evaluators and abstract machines. In *PPDP '03: Proceedings of the 5th ACM SIGPLAN international conference on Principles and practice of declaritive programming*, pages 8–19, New York, NY, USA, 2003. ACM Press.

[44] Peter J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6(4):308–320, January 1964.

[45] Olivier Danvy. A rational deconstruction of Landin's SECD machine. In *IFL*, pages 52–71, 2004.

[46] Mads Sig Ager, Olivier Danvy, and Jan Midtgaard. A functional correspondence between call-by-need evaluators and lazy abstract machines. *Information Processing Letters*, 90(5):223–232, 2004. Extended version available as the technical report BRICS-RS-04-3.

[47] John C. Reynolds. Definitional interpreters for higher-order programming languages. *Higher Order Symbol. Comput.*, 11(4):363–397, 1998.

[48] Olivier Danvy. On evaluation contexts, continuations, and the rest of the computation. Number CSR-04-1, pages 13–23, Birmingham B15 2TT, United Kingdom, 2004. Invited talk.

[49] Catherine Hope and Graham Hutton. Accurate step counting. In *Proceedings of the 17th International Workshop on the Implementation and Application of Functional Languages*, Dublin, Ireland, September 2005. To appear in the volume of selected papers from IFL 2005, to be published by Springer.

[50] Philip Wadler. Monads for functional programming. In M. Broy, editor, *Program Design Calculi: Proceedings of the 1992 Marktoberdorf International Summer School*. Springer-Verlag, 1993.

[51] Graham Hutton and Diana Fulger. Reasoning about effects: Seeing the wood through the trees. To be presented at the Symposium on Trends in Functional Programming, May 2008.

[52] Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of Haskell programs. In *ICFP*, pages 268–279, 2000.

[53] Erik Meijer. *Calculating Compilers*. PhD thesis, 1992.

[54] Don Stewart. Haskell is a strict language. `http://cgi.cse.unsw.edu.au/~dons`, 2008.

[55] Conor McBride. Functional Programming Club, University of Nottingham, January 2006. `http://sneezy.cs.nott.ac.uk/fplunch/weblog/`.

[56] Gérard Huet. The zipper. *J. Funct. Program.*, 7(5):549–554, 1997.

[57] Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In John Hughes, editor, *Proceedings of the Conference on Functional Programming and Computer Architecture*, number 523 in LNCS. Springer-Verlag, 1991.

[58] Catherine Hope and Graham Hutton. Compact Fusion. In *Proceedings of the Workshop on Mathematically Structured Functional Programming*, Kuressaare, Estonia, July 2006.

[59] Jeremy Gibbons and Graham Hutton. Proof Methods for Corecursive Programs. *Fundamenta Informaticae Special Issue on Program Transformation*, 66(4):353–366, 2005.

[60] Lambert Meertens. Calculate polytypically! In *Proceedings 8th Int. Symp. on Programming Languages: Implementations, Logics, and Programs, PLILP'96, Aachen, Germany, 24–27 Sept 1996*, volume 1140, pages 1–16. Springer-Verlag, 1996.

[61] John Launchbury. A natural semantics for lazy evaluation. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 144–154, Charleston, South Carolina, 1993.

[62] Peter Sestoft. Deriving a lazy abstract machine. *Journal of Functional Programming*, 7(3):231–264, 1997.

[63] Andy Gill. The technology behind a graphical user interface for an equational reasoning assistant. In *Proceedings of the 1995 Glasgow Workshop on Functional Programming, Electronic Workshops in Computing, Ullapool, Scotland*. Springer-Verlag, 1995.

[64] The Agda Team. The Agda Wiki, 2008. `http://appserv.cs.chalmers.se/users/ulfn/wiki/agda.php`.

[65] Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology, 2007.

[66] Shin-Cheng Mu, Hsiang-Shang Ko, and Patrik Jansson. Derivation-carrying code using dependent types. In Preparation, 2008.

[67] Nils Anders Danielsson. Derivation of an abstract machine for the language of integers and addition. Personal Communication.

[68] Gordon D. Plotkin. Call-by-name, call-by-value and the-calculus. *Theoretical Computer Science*, 1(2):125–159, 1975.

[69] Zaynah Dargaye and Xavier Leroy. Mechanized verification of CPS transformations. In *Logic for Programming, Artificial Intelligence and Reasoning, 14th Int. Conf. LPAR 2007*, volume 4790 of *Lecture Notes in Artificial Intelligence*, pages 211–225. Springer, 2007.

[70] Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. Design and correctness of program transformations based on control-flow analysis. In *TACS '01: Proceedings of the 4th International Symposium on Theoretical Aspects of Computer Software*, pages 420–447, London, UK, 2001. Springer-Verlag.

Appendix A

Instrumented abstract machine

In Chapter 5 we showed how to derive an abstract machine instrumented with space requirements for the special case of a simple language consisting of integers and addition. In this appendix we show the result of applying the same calculational process to the substitution evaluator for the full language given in section 5.5.

Instrumented substitution machine

Control stack for the abstract machine:

```
data Cont = Top | App1 Expr Cont | App2 Expr Cont
          | AppS1 Cont | AppS2 Expr Expr Int Cont
          | AppS3 Expr Cont | AbsS1 Int Cont
          | Add1 Expr Cont | Add2 Int Cont
          | AddS1 Expr Expr Int Cont | AddS2 Expr Cont
          | Cons1 Expr Cont | Cons2 Expr Cont
          | ConsS1 Expr Expr Int Cont | ConsS2 Expr Cont
          | Foldr1 Expr Expr Cont | Foldr2 Expr Expr Expr Cont
          | Foldr3 Expr Expr Cont | FoldrS1 Expr Expr Expr Int Cont
          | FoldrS2 Expr Expr Expr Int Cont | FoldrS3 Expr Expr Cont
          | Foldl1 Expr Expr Cont | Foldl2 Expr Expr Expr Cont
          | Foldl3 Expr Expr Cont | FoldlS1 Expr Expr Expr Int Cont
          | FoldlS2 Expr Expr Expr Int Cont | FoldlS3 Expr Expr Cont
```

Instrumented abstract machine:

$$
\begin{aligned}
&\mathit{spaceMach} && :: \mathit{Expr} \to \mathit{Cont} \to \mathit{MemMng} \to (\mathit{Expr}, \mathit{MemMng}) \\
&\mathit{spaceMach}\ (\mathit{Abs}\ x\ e) && c\ m = \mathit{apply'}\ c\ (\mathit{Abs}\ x\ e)\ m \\
&\mathit{spaceMach}\ (\mathit{App}\ f\ e) && c\ m = \mathit{spaceMach}\ f\ (\mathit{App1}\ e\ c)\ m \\
&\mathit{spaceMach}\ (\mathit{Add}\ x\ y) && c\ m = \mathit{spaceMach}\ x\ (\mathit{Add1}\ y\ c)\ m \\
&\mathit{spaceMach}\ (\mathit{Const}\ n) && c\ m = \mathit{apply'}\ c\ (\mathit{Const}\ n)\ m \\
&\mathit{spaceMach}\ \mathit{Nil} && c\ m = \mathit{apply'}\ c\ \mathit{Nil}\ m \\
&\mathit{spaceMach}\ (\mathit{Cons}\ x\ xs) && c\ m = \mathit{spaceMach}\ x\ (\mathit{Cons1}\ xs\ c)\ m \\
&\mathit{spaceMach}\ (\mathit{Foldr}\ f\ v\ xs) && c\ m = \mathit{spaceMach}\ xs\ (\mathit{Foldr1}\ f\ v\ c)\ m \\
&\mathit{spaceMach}\ (\mathit{Foldl}\ f\ a\ xs) && c\ m = \mathit{spaceMach}\ xs\ (\mathit{Foldl1}\ f\ a\ c)\ m
\end{aligned}
$$

Substitution function in abstract machine form:

$$
\begin{aligned}
&\mathit{subst\_s} :: \mathit{Expr} \to \mathit{Expr} \to \mathit{Int} \to \mathit{Cont} \to \mathit{MemMng} \to (\mathit{Expr}, \mathit{MemMng}) \\
&\mathit{subst\_s}\ (\mathit{Var}\ y) && v\ x\ c\ m \mid x == y = \mathit{apply'}\ c\ v\ (\mathit{free}\ 3\ m) \\
&&& \qquad\quad \mid \mathit{otherwise} = \mathit{apply'}\ c\ (\mathit{Var}\ y)\ (\mathit{free}\ (1 + \mathit{size}\ v)\ m) \\
&\mathit{subst\_s}\ (\mathit{App}\ f\ e') && v\ x\ c\ m = \mathit{subst\_s}\ f\ v\ x\ (\mathit{AppS2}\ e'\ v\ x\ c)\ (\mathit{alloc}\ (1 + \mathit{size}\ v)\ m) \\
&\mathit{subst\_s}\ (\mathit{Abs}\ y\ e) && v\ x\ c\ m \mid x == y = \mathit{apply'}\ c\ (\mathit{Abs}\ y\ e)\ (\mathit{free}\ (1 + \mathit{size}\ v)\ m) \\
&&& \qquad\quad \mid \mathit{otherwise} = \mathit{subst\_s}\ e'\ v\ x\ (\mathit{AbsS1}\ y'\ c)\ m \\
&\quad \textbf{where}\ (y', e') = \textbf{if}\ (\neg\ (\mathit{elem}\ y\ (\mathit{freevars}\ v))) \\
&&& \qquad\quad \textbf{then}\ (y, e) \\
&&& \qquad\quad \textbf{else}\ (\mathit{fresh}\ (\mathit{freevars}\ e +\!\!+ \mathit{freevars}\ v), \mathit{subst}\ e\ (\mathit{Var}\ y')\ y) \\
&\mathit{subst\_s}\ (\mathit{Add}\ x'\ y) && v\ x\ c\ m = \mathit{subst\_s}\ x'\ v\ x\ (\mathit{AddS1}\ y\ v\ x\ c)\ (\mathit{alloc}\ (1 + \mathit{size}\ v)\ m) \\
&\mathit{subst\_s}\ (\mathit{Const}\ n) && v\ x\ c\ m = \mathit{apply'}\ c\ (\mathit{Const}\ n)\ (\mathit{free}\ (1 + \mathit{size}\ v)\ m) \\
&\mathit{subst\_s}\ \mathit{Nil} && v\ x\ c\ m = \mathit{apply'}\ c\ \mathit{Nil}\ (\mathit{free}\ (1 + \mathit{size}\ v)\ m) \\
&\mathit{subst\_s}\ (\mathit{Cons}\ x'\ xs) && v\ x\ c\ m = \mathit{subst\_s}\ x'\ v\ x\ (\mathit{ConsS1}\ xs\ v\ x\ c)\ (\mathit{alloc}\ (1 + \mathit{size}\ v)\ m) \\
&\mathit{subst\_s}\ (\mathit{Foldr}\ f\ v'\ xs) && v\ x\ c\ m = \mathit{subst\_s}\ f\ v\ x\ (\mathit{FoldrS1}\ v'\ xs\ v\ x\ c)\ (\mathit{alloc}\ (1 + \mathit{size}\ v)\ m) \\
&\mathit{subst\_s}\ (\mathit{Foldl}\ f\ a\ xs) && v\ x\ c\ m = \mathit{subst\_s}\ f\ v\ x\ (\mathit{FoldlS1}\ a\ xs\ v\ x\ c)\ (\mathit{alloc}\ (1 + \mathit{size}\ v)\ m)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{apply'} && :: \mathit{Cont} \to \mathit{Expr} \to \mathit{MemMng} \to (\mathit{Expr}, \mathit{MemMng}) \\
&\mathit{apply'}\ \mathit{Top} && v\ \ m = (v, \mathit{free}\ 1\ m) \\
&\mathit{apply'}\ (\mathit{App1}\ e\ c) && f\ \ m = \mathit{spaceMach}\ e\ (\mathit{App2}\ f\ c)\ m \\
&\mathit{apply'}\ (\mathit{App2}\ (\mathit{Abs}\ x\ e')\ c) && v\ \ m = \mathit{subst\_s}\ e'\ v\ x\ (\mathit{AppS1}\ c)\ (\mathit{free}\ 1\ m)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{apply'}\ (\mathit{Add1}\ y\ c)\ (\mathit{Const}\ n)\ \ m = \mathit{spaceMach}\ y\ (\mathit{Add2}\ n\ c)\ (\mathit{free}\ 1\ m)\\
&\mathit{apply'}\ (\mathit{Add2}\ n\ c)\ (\mathit{Const}\ m')\ m = \mathit{apply'}\ c\ (\mathit{Const}\ (n + m'))\ (\mathit{free}\ 2\ m)\\
&\mathit{apply'}\ (\mathit{Cons1}\ xs\ c)\ \ \ \ \ \ \ \ v\ \ m = \mathit{spaceMach}\ xs\ (\mathit{Cons2}\ v\ c)\ m\\
&\mathit{apply'}\ (\mathit{Cons2}\ v\ c)\ \ \ \ \ \ \ \ vs\ m = \mathit{apply'}\ c\ (\mathit{Cons}\ v\ vs)\ m\\
&\mathit{apply'}\ (\mathit{Foldr1}\ f\ v\ c)\ \ \ \ \ \ vs\ m = \textbf{case}\ vs\ \textbf{of}\\
&\quad \mathit{Nil} \rightarrow \mathit{spaceMach}\ v\ c\ (\mathit{free}\ (2 + \mathit{size}\ f)\ m)\\
&\quad (\mathit{Cons}\ y\ ys) \rightarrow \mathit{spaceMach}\ f\ (\mathit{Foldr2}\ v\ y\ ys\ c)\ (\mathit{free}\ 1\ m)\\
&\mathit{apply'}\ (\mathit{Foldr2}\ v\ y\ ys\ c)\ \ \ \ f'\ \ m =\\
&\quad \mathit{spaceMach}\ (\mathit{Foldr}\ f'\ v\ ys)\ (\mathit{Foldr3}\ f'\ y\ c)\ (\mathit{alloc}\ (1 + \mathit{size}\ f')\ m)\\
&\mathit{apply'}\ (\mathit{Foldr3}\ f\ y\ c)\ \ \ \ \ \ z\ \ m = \mathit{spaceMach}\ (\mathit{App}\ (\mathit{App}\ f\ y)\ z)\ c\ (\mathit{alloc}\ 1\ m)\\
&\mathit{apply'}\ (\mathit{Foldl1}\ f\ a\ c)\ \ \ \ \ \ vs\ m = \textbf{case}\ vs\ \textbf{of}\\
&\quad \mathit{Nil} \rightarrow \mathit{spaceMach}\ a\ c\ (\mathit{free}\ (2 + \mathit{size}\ f)\ m)\\
&\quad (\mathit{Cons}\ y\ ys) \rightarrow \mathit{spaceMach}\ f\ (\mathit{Foldl2}\ a\ y\ ys\ c)\ (\mathit{free}\ 1\ m)\\
&\mathit{apply'}\ (\mathit{Foldl2}\ a\ y\ ys\ c)\ \ \ \ f'\ \ m =\\
&\quad \mathit{spaceMach}\ (\mathit{App}\ (\mathit{App}\ f'\ a)\ y)\ (\mathit{Foldl3}\ f'\ ys\ c)\ (\mathit{alloc}\ (2 + \mathit{size}\ f')\ m)\\
&\mathit{apply'}\ (\mathit{Foldl3}\ f\ ys\ c)\ \ \ \ \ \ a'\ \ m = \mathit{spaceMach}\ (\mathit{Foldl}\ f\ a'\ ys)\ c\ m
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{apply'}\ (\mathit{AppS1}\ c)\ \ \ \ \ \ \ \ \ \ \ e\ \ m = \mathit{spaceMach}\ e\ c\ (\mathit{free}\ 1\ m)\\
&\mathit{apply'}\ (\mathit{AppS2}\ e'\ v\ x\ c)\ \ \ \ \ f\ \ m = \mathit{subst\_s}\ e'\ v\ x\ (\mathit{AppS3}\ f\ c)\ m\\
&\mathit{apply'}\ (\mathit{AppS3}\ f'\ c)\ \ \ \ \ \ \ \ e\ \ m = \mathit{apply'}\ c\ (\mathit{App}\ f'\ e)\ m\\
&\mathit{apply'}\ (\mathit{AbsS1}\ y\ c)\ \ \ \ \ \ \ \ e\ \ m = \mathit{apply'}\ c\ (\mathit{Abs}\ y\ e)\ m\\
&\mathit{apply'}\ (\mathit{AddS1}\ y\ v\ x\ c)\ \ \ \ \ x'\ \ m = \mathit{subst\_s}\ y\ v\ x\ (\mathit{AddS2}\ x'\ c)\ m\\
&\mathit{apply'}\ (\mathit{AddS2}\ x''\ c)\ \ \ \ \ \ \ y'\ \ m = \mathit{apply'}\ c\ (\mathit{Add}\ x''\ y')\ m\\
&\mathit{apply'}\ (\mathit{ConsS1}\ xs\ v\ x\ c)\ \ \ \ y\ \ m = \mathit{subst\_s}\ xs\ v\ x\ (\mathit{ConsS2}\ y\ c)\ m\\
&\mathit{apply'}\ (\mathit{ConsS2}\ y\ c)\ \ \ \ \ \ \ \ ys\ m = \mathit{apply'}\ c\ (\mathit{Cons}\ y\ ys)\ m\\
&\mathit{apply'}\ (\mathit{FoldrS1}\ v'\ xs\ v\ x\ c)\ f'\ \ m =\\
&\quad \mathit{subst\_s}\ v'\ v\ x\ (\mathit{FoldrS2}\ f'\ xs\ v\ x\ c)\ (\mathit{alloc}\ (1 + \mathit{size}\ v)\ m)\\
&\mathit{apply'}\ (\mathit{FoldrS2}\ f'\ xs\ v\ x\ c)\ v'\ \ m = \mathit{subst\_s}\ xs\ v\ x\ (\mathit{FoldrS3}\ f'\ v'\ c)\ m\\
&\mathit{apply'}\ (\mathit{FoldrS3}\ f'\ v''\ c)\ \ \ \ \ ys\ m = \mathit{apply'}\ c\ (\mathit{Foldr}\ f'\ v''\ ys)\ m\\
&\mathit{apply'}\ (\mathit{FoldlS1}\ a\ xs\ v\ x\ c)\ \ f'\ \ m =\\
&\quad \mathit{subst\_s}\ a\ v\ x\ (\mathit{FoldlS2}\ f'\ xs\ v\ x\ c)\ (\mathit{alloc}\ (1 + \mathit{size}\ v)\ m)\\
&\mathit{apply'}\ (\mathit{FoldlS2}\ f'\ xs\ v\ x\ c)\ a'\ \ m = \mathit{subst\_s}\ xs\ v\ x\ (\mathit{FoldlS3}\ f'\ a'\ c)\ m\\
&\mathit{apply'}\ (\mathit{FoldlS3}\ f'\ a'\ c)\ \ \ \ \ ys\ m = \mathit{apply'}\ c\ (\mathit{Foldl}\ f'\ a'\ ys)\ m
\end{aligned}
$$

Size of an expression:

```
class Sized a where
  size :: a → Int

instance Sized Expr where
  size (Var x) = 1 + size x
  size (Abs x e) = 1 + size x + size e
  size (App f e) = 1 + size f + size e
  size (Add x y) = 1 + size x + size y
  size (Const n) = 1 + size n
  size Nil = 1
  size (Cons x xs) = 1 + size x + size xs
  size (Foldr f v xs) = 1 + size f + size v + size xs
  size (Foldl f a xs) = 1 + size f + size a + size xs

instance Sized Int where
  size n = 1
```