Shape Analysis

Milan Sekanina

August 1998

Ph.D. Thesis

University of Technology, Sydney

Certificate

I certify that this thesis has not already been submitted for any degree and is not being submitted as part of candidature for any other degree.

I also certify that the thesis has been written by me and that any help that I have received in preparing this thesis, and all sources used, have been acknowledged in this thesis.

Milan Sekanina

Acknowledgments

The following is an incomplete list of the people whom I owe a debt of gratitude for helping me produce this thesis:

My supervisor, Barry Jay, who not only introduced me to shape theory, helped me push my research along and taught me all I know about about scientific writing, but went way beyond the call of his supervisorial duty by trying to make my stay in Australia as trouble-free as possible.

My second supervisor, Jenny Edwards for patiently reading and correcting numerous drafts of my thesis.

All the other people who through the years worked on shape at UTS and helped me with their suggestions, especially Paul Steckler, Dave Clarke, and Daniel Mahler.

The referees, including Chris Hankin and Eugenio Moggi, for their valuable suggestions. E Moggi in particular had a major impact on the final structure of the thesis by suggesting a shift in perspective which led to its considerable simplification.

And my family and friends for bearing with me when the going got harder.

Contents

1 Introduction **1**
1.1 Semantics of shape 3
1.2 Shape analysis 5
1.2.1 Shape analysis and abstract interpretation 9
1.2.2 Shape analysis and data flow analysis 10
1.2.3 Shape analysis and partial evaluation 11
1.2.4 Shape analysis and two-level languages 12
1.2.5 Shape analysis and dependent types 12
1.3 Structure of the thesis 13

2 The SIZE language **15**
2.1 Typed Lambda Calculi 15
2.1.1 Types and terms 16
2.1.2 Manipulating terms 19
2.1.3 Terms and equality 25
2.1.4 Evaluating terms 28
2.2 Extending the calculus 31
2.2.1 Combinators 31
2.2.2 Product type 32
2.2.3 Unit 33
2.2.4 Natural numbers 33

2.2.5 Booleans and conditionals 34
2.2.6 Equality 35
2.2.7 Recursion 35
2.2.8 Errors 37
2.3 The Size language 37

3 The Vec language **51**
3.1 Vectors and Arrays 51
3.2 The Vec language 53
3.3 Expressive power 60
3.3.1 Array indexing 60
3.3.2 Second order vector operations 62
3.3.3 Linear algebra 63

4 Shape analysis **65**
4.1 The shape translation 66
4.2 Examples 69
4.2.1 Vectors 69
4.2.2 Array indexing 70
4.2.3 Second order vector operations 70
4.2.4 Linear algebra 71
4.3 Properties of shape analysis 71

5 Simplifying shapes **77**
5.1 Size with checks 78
5.2 Shape analysis with checks 83

6 Extending shape analysis **88**
6.1 Shapely operations 88
6.2 Data conditionals 91
6.3 Data polymorphism 94

6.4 Shape polymorphism 95

7 Conclusions 97

A Proofs of confluence 107

A.1 Confluence of Size 107

A.2 Confluence of Vec 112

B Implementation 114

B.1 Shape analysis to Size 114

B.2 Shape analysis to Size_{C} 115

List of Figures

2.1 Typing judgments for the simply typed lambda calculus 18
2.2 Eager operational semantics of the simply typed calculus . . . 29
2.3 SIZE combinators and their types 39
2.4 Typing judgments for SIZE 39
2.5 Base reductions for SIZE 40
2.6 Operational semantics of SIZE 42

3.1 Typing of VEC combinators 54
3.2 Typing judgments for VEC 56
3.3 Base reductions for VEC 57
3.4 Operational semantics of VEC– Part 1 57
3.5 Operational semantics of VEC– Part 2 58

4.1 Shape translation of terms 67

5.1 SIZE_C combinators and their types 78
5.2 Base reductions for SIZE_C 80
5.3 Equality reductions in SIZE_C 81
5.4 Operational semantics of SIZE_C 82
5.5 Shape analysis in SIZE_C 84

Abstract

Most data structures commonly supported by programming languages, including arrays, lists and trees, can be split into two components – the underlying structure (or the shape) and the data stored within. Though the benefits of manipulating the data alone have been known for a long time (as witnessed, for example, by the wide-spread use of data polymorphism), only in recent years has greater attention been paid to the shape content as well, primarily by groups studying areas such as intensional polymorphism or polytypism. Shape theory, a part of this programme of work, unifies the various notions of shape under a single framework.

This thesis studies shape analysis, a branch of shape theory which extracts the shapes of data structures and uses them for program optimisation. It concentrates on using shape analysis for detecting errors arising from ill formed or incompatible shapes. A typical example of such shape errors might be multiplying matrices of ill matched dimensions. Since shape analysis ignores all data and data-based computations, it has a potential to be very efficient, as well as completely safe error-checking method.

As a vehicle for this study, Vec, a simply typed lambda calculus supporting a vector type constructor (and thus arbitrary arrays), and its sublanguage of shapes, Size, are introduced. Two kinds of shape analysis of Vec are then defined. One maps a Vec term to its shape in Size, ignoring in the process all data computations. Important properties of this shape translation are then proved, especially its ability to detect all shape errors in a Vec term. Later a different kind of shape analysis is introduced, one suitable for a restricted set of Vec terms and producing very simple shapes for some higher order terms.

Chapter 1

Introduction

The use of the word "shape" is a common and familiar sight in phrases such as "the shape of an array" (referring to an array's dimensions), "the shape of a heap" (referring to the form of a heap-allocated data structure) or "the shape of a system of linear equations" (referring to the pattern of non-zero coefficients of such a system). These (and similar) examples form a very diverse collection and, at the first glance, the only thing they seem to share is that in each case the word "shape" refers to something vaguely endowed with a physical form – a rectangular array, a triangular matrix, a circular list. However, a deeper investigation reveals that they have more in common – in all the examples, "the shape of a data structure" refers to a component of the data structure, a component that can be looked upon as some underlying structure with holes for storing the rest of the data structure, e.g. the array's entries, the storage for the heap. One can thus separate the data structure into two parts – its shape and the data stored within (since the number of the data is typically finite, we can view it as a list). Furthermore, any data entry can be stored in any hole in the shape, and any such assignment produces a well-formed data structure. Thus a matrix is given by a pair of sizes and a list of entries (ordered in some implicit way), a finite dimensional array is determined by a list of sizes and a list of entries, a labeled tree is given by an unlabeled one and a list of labels.

Of course, similar observations (though in isolated cases only) have been made long ago and shapes have been used in some form or another since the advent of programming languages. Arrays in some early languages such

as APL [Iverson, 1962] are constructed from a list of sizes and a list of entries. Algorithms often use shape information for program optimisation. However, until recently there has been no general theory unifying the various notions of shape under a single framework. There have, of course, been theories powerful enough to describe shape, such as dependent type theory [Martin-Löf, 1984], but their focus lies more in areas such as theorem proving than programming language design. Only in the last few years have language design techniques treating and using shape (or its close approximations) in a more general and uniform manner emerged. Some of this research falls under the framework of the Bird-Meertens formalism [Meijer et al., 1991] – such as the work on polytypic operations [Jeuring and Jansson, 1996], operations defined by induction on the structure of a datatype and thus applicable to values of different types (and shapes). The generic theory of datatypes developed by [Hoogendijk, 1997] has similar goals. Intensional polymorphism [Harper and Morrisett, 1995] uses a type-based run time analysis to generate efficient code. Shape theory, a recent addition to the extensive suite of language design techniques, unifies the various notions of shape under a single, general framework.

The basic concept of shape theory, one capturing the core characteristics of shapes in the above examples, is that of shapely datatype, a concept originally coming from categorical semantics of programming languages. Shape theory and its implications for programming language design are now being intensely studied (primarily by the Algorithms and Languages Group at University of Technology in Sydney) but many potential applications remain largely unexplored. The two branches of shape theory studied so far are shape polymorphism and shape analysis. These are, up to a point, orthogonal methods, but their underlying principles remain the same – they both strive to shift the responsibility for shape computation onto the compiler, thus, in effect, turning shapes into compile time entities. We will talk about them and their implications for language design in a greater detail shortly, but before we do, it might be useful to present the basic concepts of the categorical semantics of shape as it helps to understand our motivations. However, an effort has been made to keep the exposure to category theory to a minimum.

1.1 Semantics of shape

This section will introduce the theoretical concepts underlying shape theory. The full details can be found in [Jay, 1995] or [Jay, 1994]. The category theory needed is elementary; the definitions of all the notions unexplained here can be found in any introductory text on category theory such as [Barr and Wells, 1990] or [Asperti and Longo, 1991].

In a categorical semantics, types are typically represented by objects and terms by arrows in a suitable category $\mathcal{C}$. Datatype *constructors*, such as the matrix type constructor, then correspond to functors over $\mathcal{C}$. So let us assume we have a functor $F : \mathcal{C} \to \mathcal{C}$ and an object σ such that $F\ \sigma$ represents a type. The operations extracting the data list and the shape from $F\ \sigma$ (when they exist) have the form

$$\begin{aligned} \mathtt{data} &: F\ \sigma \to L\ \sigma \\ \mathtt{shape} &: F\ \sigma \to \mathtt{shapes} \end{aligned}$$

with `shapes` being the type of shapes of $F\ \sigma$ and $L : \mathcal{C} \to \mathcal{C}$ the list functor.

Conversely, any list of σ's together with a shape determine a unique structure of type $F\ \sigma$, provided the number of "holes" in the shape (its *arity*) matches the length of the list. This is just another way of saying that the following square is a pullback:

$$\begin{array}{ccc} F\ \sigma & \xrightarrow{\mathtt{data}} & L\ \sigma \\ {\scriptstyle \mathtt{shape}}\downarrow & \lrcorner & \downarrow{\scriptstyle \mathtt{length}} \\ \mathtt{shapes} & \xrightarrow[\mathtt{arity}]{} & \mathtt{nat} \end{array}$$

(where `nat` is the natural numbers object). The above diagram exactly captures the data/shape partitioning of $F\ \sigma$.

For a concrete example, let us consider matrices of integers, with the

matrix functor being $M : \mathcal{C} \to \mathcal{C}$. The pullback then has the following form

$$\begin{array}{ccc} M\ \mathsf{int} & \xrightarrow{\mathtt{data}} & L\ \mathsf{int} \\ \mathtt{shape}\downarrow & \lrcorner & \downarrow\mathtt{length} \\ \mathsf{nat} \times \mathsf{nat} & \xrightarrow[*]{} & \mathsf{nat} \end{array}$$

with `*` representing the multiplication of natural numbers.

The challenge now is to describe the `shapes` object and the operations `shape` and `arity` in a general way. To do that, we make several observations. Firstly, natural numbers can be viewed as lists of units (and vice versa). This is reflected in the fact that, in any suitable category, the objects nat and L 1 are isomorphic (where 1 is the terminal object, an object with a single value, thus carrying no information). Secondly, the shape of a data structure can be thought of as the structure itself with all the data replaced by units (representing the holes). Thus `shapes` should be F 1 and the `shape` arrow is just F ! (where ! is the unique arrow into the terminal object). This is consistent with the matrix case above since the objects nat $\times$ nat and M 1 are isomorphic. Finally, the operation `arity` can be thought of simply as another `data` operation, returning the list of "data" in the shape, with the data now being the terminal objects. When we apply these observations to the pullback diagram above, we get the following diagram

$$\begin{array}{ccc} F\ \mathsf{int} & \xrightarrow{\mathtt{data}_{\mathsf{int}}} & L\ \mathsf{int} \\ F\ !\downarrow & \lrcorner & \downarrow L\ ! \\ F\ 1 & \xrightarrow[\mathtt{data}_1]{} & L\ 1 \end{array}$$

Functors for which there exists a family of such arrows `data` (satisfying some naturality conditions) are *shapely over lists* [Jay, 1995]. Thus such functors correspond to those datatype constructors that can be split into the shape and data in the way discussed above. Of course, not all type constructors are

shapely – function types, for example, are not typically shapely, and neither are types of sets.

1.2 Shape analysis

An important branch of shape theory, one that has been intensely studied, is shape polymorphism [Jay and Cockett, 1994]. Shape polymorphism allows a function to be uniformly applied to structures with various shapes. Let us consider, for example, mapping a function across a data structure. Whether a function is being mapped over a list or a tree, the overall specification remains the same – go through all the data and apply the function to each. The challenge is then to uniformly describe the type and/or the algorithm for map. In a shape polymorphic type system, the type of map (and the algorithm for its application) is parametric in the choice of the (shapely) datatype constructor, thus covering all the cases mentioned above. The FML language (standing for Functorial ML) is an extension of ML supporting shape polymorphism [Bellè et al., 1996].

Evaluating shapes and using the derived shape information before accessing any data is the domain of shape analysis, another branch of shape theory, one with perhaps even greater potential benefits than shape polymorphism. Shape analysis has not yet been extensively studied and this thesis will try to contribute to the understanding of the processes involved.

When it is possible to extract the shape prior to the actual data computation, one can then profit from the early use of the shape information. The potential benefits are manifold, ranging from speeding up the program's performance and improved memory allocation to error detection. Error detection will be the main focus of this thesis, but let us first illustrate the practical importance of shape-based optimisations. Consider, for example, the problem of multiplying several matrices. Since matrix multiplication is associative, the chosen order of multiplication does not affect the result, but it does have a great impact on the number of (integer) multiplications needed. This *optimal matrix-parenthesization problem* [Cormen et al., 1990] is a shape-based optimisation which is an essential part of any efficient matrix multiplication algorithm. Many other examples can be found – solving a sparse system of linear equations often involves reordering the system so as

to minimise the number of non-zero coefficients appearing during the solving phase, an optimisation that is based purely on the sparsity pattern, i.e. the shape, of the system [Duff et al., 1986]. In practice, many such algorithms (and in particular both the examples mentioned above) are optimised with respect to one shape and then applied to several systems with the same shape but different data, thus further stressing the significance of such shape-based optimisations.

Shape analysis might also have a big impact in the world of parallel computing, primarily in areas such as load balancing, since both the sizes of data structures and the associated communication patterns often depend solely on shapes. The finite element method for solving partial differential equations over a discretised domain is a case in point – when dividing the task across processors, one has to carve a graph representing the domain into pieces of similar size while cutting through as few edges (representing the necessary communication links) as possible [Kumar et al., 1994]. Another important application is deriving static estimates of program execution costs. Execution costs are often primarily shape-dependent since the bulk of the execution time of, say, an array program depends on the sizes of the array, not on its entries, and consequently a number of cost estimating techniques relying on shape information have been developed. Skillicorn's cost calculus for a skeleton-based language [Skillicorn, 1994] is an example of such (informal) use of shape. Inferring the lengths (i.e. the shapes) of vectors is the basis for size inference [Blelloch et al., 1991], a technique used for deriving efficient parallel implementations of NESL [Blelloch, 1992]. Similar estimates of variable sizes are used in [Ching, 1986] for compiling APL.

We have used shape analysis similar to the one presented in this thesis to infer static estimates of execution costs of array based programs in the PRAM setting [Jay et al., 1997]. The system has been implemented and shown to provide accurate estimates.

FISh, an Algol-like language heavily relying on the ideas of shape is currently being developed [Jay and Steckler, 1998]. FISh supports a high-level programming style similar to that typical of functional languages while using shape analysis to generate very fast and memory-efficient code. For example, shape analysis in FISh is used to avoid unnecessary boxing of array entries.

Let us now consider error detection. This has, of course, long been an intensely studied area. Introduction of types and automatic type checking

(and type inference) have been largely motivated by the desire to eliminate certain kinds of errors (type errors). In a similar vein, we propose shape analysis as a means for identifying *shape errors* [Jay and Sekanina, 1997], errors arising from working with ill-formed or incompatible shapes. Examples include zipping together two lists of different lengths or multiplying matrices of ill-matched sizes. Perhaps even more importantly, array access errors, i.e. accessing an out-of-bounds array entry, fall into the same category.

For shape analysis to be successful, it is necessary for the analysed operations to satisfy an important restriction: all shapes involved must depend solely on the shapes of the inputs (they have to be *shapely operations*). Thus shape analysis can analyse (and detect errors in), say, matrix multiplication – here the sizes of the resulting matrix are determined by the sizes of the two input matrices. On the other hand, shape analysis cannot handle operations such as filtering of a list, where the length of the result depends not only on the length (the shape) of the input but also on its entries (data). The potential significance of shape analysis thus depends on the proportion of shapely operations in real-world computing. Fortunately, it turns out that many algorithms and operations used in practice, particularly in areas such as array-based computations and linear algebra, are, indeed, shapely – examples include operations such as the scalar multiplication of vectors and Gaussian elimination. However, shape analysis can be of use even when shapely operations are intermixed with non-shapely ones – in such cases one could interleave shape analysis of the shapely parts with direct evaluation of the rest, as suggested in [Jay, 1996]. This topic will not be addressed in this thesis, though.

We will study shape analysis primarily in the context of arrays and array-based computations. As a vehicle for this study, we introduce VEC, a simple language supporting vectors. The overriding concern in its design was to ensure that shape analysis is able to analyse all VEC programs, and therefore that only shapely operations are expressible in it. VEC is based on the simply typed lambda calculus, with its type system given by

$$\begin{array}{rcl} \delta & ::= & \mathsf{nat} \mid \mathsf{bool} \mid \ \ldots \\ \tau & ::= & \delta \mid \mathsf{un} \mid \mathsf{sz} \mid \tau \times \tau \mid \mathsf{vec}\ \tau \\ \sigma & ::= & \tau \mid \sigma \times \sigma \mid \sigma \to \sigma \ . \end{array}$$

Here δ ranges over *datum* types such as natural numbers and booleans, types

having no (or, rather, trivial) shape content. The stratification between *data* types τ and *phrase* types σ prevents formation of types such as vectors of functions which are undesirable for technical reasons – rather than storing terms of such types sequentially in computer memory (as one would other vectors and arrays), one would have to employ something like arrays of pointers which are both slow and difficult to analyze. Finite dimensional arrays can be represented in VEC by nesting the vector type constructor vec – shape analysis will ensure that all entries in a vector have the same length and that nested vectors are thus, indeed, arrays. The emphasis on shapely operations has lead to some novel design decisions such as the introduction of the type of *sizes* sz, the shape equivalent of natural numbers used as vector lengths. Shape errors, such as array access errors, are made explicit in the syntax of VEC by introducing the error combinator err.

The shapes of VEC terms are isolated in its sublanguage SIZE. The SIZE type system is simpler than that of VEC, as it involves neither datum nor vector types:

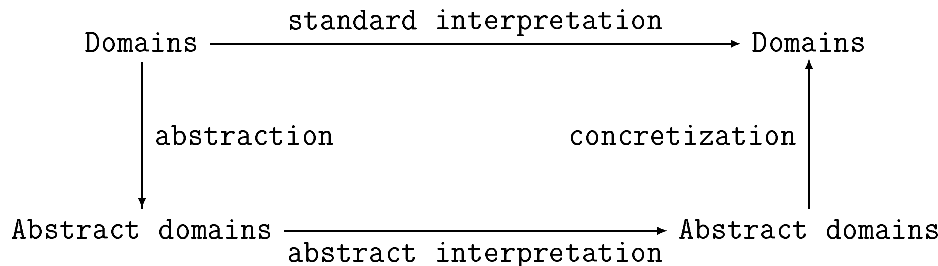
$$\sigma ::= \mathsf{un} \mid \mathsf{sz} \mid \sigma \times \sigma \mid \sigma \rightarrow \sigma \ .$$

We will study two kinds of shape analysis on VEC, each suited to a different kind of problem. First we introduce a very general analysis, applicable to a wide range of programs, as a mapping from terms in VEC to their shapes in SIZE. We will show that this shape analysis does indeed produce the shapes it should and, in particular, that it detects all shape errors (including array access errors) in a VEC program. Later we introduce a different kind of shape analysis, suitable for a restricted set of VEC programs that produces very simple shapes of some higher-order terms.

Since shape is a fairly simple and ubiquitous notion, it comes as no surprise that there is a considerable amount of related material, though none is as general as what we propose. Shape analysis itself is closely related to several existing methods such as abstract interpretation or partial evaluation. The rest of this section will thus discuss the relationships between shape analysis and various existing methods, namely abstract interpretation, data flow analysis, partial evaluation, two-level languages and dependent types.

1.2.1 Shape analysis and abstract interpretation

Abstract interpretation [Cousot and Cousot, 1979] is a semantics-based method of static program analysis that studies the (at compile time typically uncomputable) run-time behaviour of programs by concentrating only on the (statically computable) properties of interest. The general mechanism is captured in the following diagram



(here `Domains` and `Abstract domains` are generic terms representing the suitable semantics which may vary according to the type of the analysed program, be it functional, declarative, parallel or so on). Thus one *abstracts* from the standard semantics of a program, analyses it using abstract interpretation and then gets back the concrete representation. As the abstract representation does not capture the whole of the standard semantics, the above diagram does not quite commute – the requirement is for the analysis to be "safe", that is the concretised abstract domain has to be "at least as general" as the original one (again, the formal definitions vary). A detailed description of abstract interpretation can be found in, for example, [Abramsky and Hankin (editors), 1987] or [Jones and Nielson, 1992].

Abstract interpretation provides a very general framework for various program analyses and, up to a point, we can view shape analysis in this light as well – in our case `abstraction` would correspond to shape extraction, `abstract interpretation` to shape manipulation (computation of the resulting shapes and detection of eventual errors) and `concretization` would map a shape to the set of terms with this shape. Even the safety condition has an analog in the main claim about shape analysis (stated in Theorem 4.3.4) which says that shape analysis of a program detects all shape errors, but may "detect an error" when in fact there is none.

Abstract interpretation (together with data flow analysis discussed below) is the framework for an extensive body of work on array access checking, an area of particular interest to us. Since, in general, it is not possible to detect at compile-time whether all array accesses are within bounds, run-time checks have to be performed (or one runs the risk of corrupting the memory). In programs involving a large number of array accesses, these checks can considerably slow down the program's execution and devising efficient optimisation techniques for such programs thus becomes essential. Abstract interpretation based techniques concentrate on estimating the values integer variables can take (as these are used as array indices) and use these estimates to identify the redundant checks. Typically these potential values are expressed as intervals (or ranges, hence range check optimization – see, for example, [Cousot and Halbwachs, 1978]). The inference rules vary in their sophistication as one has to balance the benefits resulting from more complex analyses against the higher costs these incur in terms of spent computing time and power. Thus analyses range from the relatively simple ones [Harrison, 1977] to ones that use full-blown theorem provers (such as [Suzuki and Ishihata, 1977]) and are thus probably too complicated for practical use. Some of the simpler analyses have long been used in optimising compilers – see, for example, [Markstein et al., 1982] for the implementation of array range checker in the IBM PL.8 compiler.

It is important to note that shape analysis does not directly compete with these methods as its premises are slightly different. We have already said that we will treat shapes as information available to us during shape analysis, something that typically is not true of the methods described above. As well, our restriction to shapely programs will allow us to detect *all* array access errors.

1.2.2 Shape analysis and data flow analysis

Data flow analysis [Aho et al., 1986] is a generic term covering a wide range of program-transformation techniques such as constant folding or dead-code elimination. The common theme of all these techniques is that they base their transformations on information about the data (and control) flow of a program. Data-flow based optimising techniques can be found in every optimising compiler. There is an enormous body of work on data-flow analysis and

a considerable proportion of it can be viewed as working with shape in some form (especially with the shapes of arrays, as array programs are, understandably, the subject of much attention – see, for example, [Feautrier, 1991]).

An important application is data-flow analysis of array access checks. A number of such techniques has been studied (a useful comparison of the benefits of various optimising techniques can be found in [Kolte and Wolfe, 1995]). The most commonly used techniques (and those most beneficial) are check elimination and propagation of checks out of loops [Gupta, 1993]. Other transformation techniques, more expensive but potentially producing better optimizations, such as conservative expression substitution and loop guard elimination, are described in [Asuru, 1992].

Understandably, techniques working with shapes of arrays are much more common than those working with more complex shapes. Pointer analysis of heap-allocated storage (also known as "shape analysis" [Sagiv et al., 1996]) is a rare example of the latter. It uses data flow-based analysis to estimate the shapes (represented by pointers) heap-allocated data structures can take. Its goal is not primarily to estimate the sizes of these shapes, but rather the form they can take – be it a list, a circular list, a tree or so on. These estimates are then used for generating efficient implementations, both sequential and parallel.

1.2.3 Shape analysis and partial evaluation

Partial evaluation [Jones et al., 1993] is a program transformation technique that specialises programs with respect to some (or some parts of) their inputs. Understandably, such a specialisation can significantly speed up the program's execution. Its applications are manifold, ranging from compiler generation to pattern matching.

Shape analysis can also be seen as a special kind of partial evaluation, one where programs are specialised with respect to the shapes of their inputs. To make this specialisation possible, it has to be possible to separate, at the syntactic level, shapes and data. Moreover the parts of the program that are only shape-dependent have to be identified (a form of binding time analysis). As we have already mentioned, in this thesis we will concentrate on shape analysis of shapely programs, that is programs where all shapes can

be determined once the input shapes are given. On the other hand, shape analysis will not try to evaluate any data, even though some may also be only shape-dependent (and would thus probably be evaluated by standard partial evaluation techniques).

We expect that some of the major practical applications of shape analysis will be in the area of partial evaluation. For example, we have already described several algorithms (matrix parenthesization, finite element method) that use shape information for optimisations, and such optimisations are again a form of shape-based partial evaluation. Partial evaluation has already been used for similar tasks – specialising an algorithm solving sparse systems of linear equations to a given sparsity pattern [Gustavson et al., 1970], or specialising a ray-tracing program to a given scene [Mogensen, 1986].

1.2.4 Shape analysis and two-level languages

Partial evaluation is closely linked to the theory of two (and higher) level languages [Nielson and Nielson, 1992]. In such a language, terms are annotated with additional binding time information which, in two-level languages, corresponds to the distinction between compile-time (static) and run-time (dynamic) entities. Not every binding time annotation is, of course, valid – for example, a run-time function should not return a compile-time value. The rules ensuring valid annotations are the equivalent of binding time analysis in partial evaluation.

Again, shape analysis is closely connected with the theory of two-level languages. Since shape analysis computes shapes prior to working with the data, shapes can be viewed as the static, and data as the dynamic entities in a language. A two-level language supporting the shapes of nested sequences (a generalisation of arrays) was introduced in [Bellè and Moggi, 1997] and used for shape analysis of the Nested Sequence Calculus [Suciu and Tannen, 1994].

1.2.5 Shape analysis and dependent types

A dependent type [Martin-Löf, 1984] is a type whose value depends on the value of a term. Such type systems are very powerful – one can express, say,

the type of vectors of length at least n as

$$\mathsf{vec}_{\geq n}\ X = \sum_{\forall m \geq n} X^m$$

that is, as the sum of all vectors of length at least n. Dependent types blur the distinction between compile time and run time as types (typically compile time entities) may depend on the result of a term (run time) computation. One of the consequences is that, in general, type checking (and type inference) for such systems is undecidable.

We can formulate shape analysis as the type checking problem for a suitable dependent type system. Let us recall that, for any shapely type constructor F we have a type of shapes S (given by the object $F\ 1$ from the pullback diagram from Section 1.1). Then a type $F\ X$ can be expressed as

$$F\ X = \sum_{\forall s:S} X^{\mathtt{arity}(s)}\ .$$

In other words, $F\ X$ is the sum of $\mathtt{arity}(s)$ copies of X (representing the data) over all shapes s. Shape analysis then becomes equivalent to type checking for such a system. As we have already said, in general this is an undecidable problem and the challenge is therefore to come up with a system powerful enough to be interesting but allowing static type checking. These points can be reconciled and there is a number of languages based on dependent types (see, for example, [Bellè and Moggi, 1997]).

Closely related to the work presented in this thesis is the research outlined in [Xi and Pfenning, 1997], where an extension of ML with a restricted form of dependent types allowing static type checking is used for eliminating array bound checks. This research is still in its early stages and detailed results are not yet available.

1.3 Structure of the thesis

The structure of the thesis is as follows:

- In the following chapter we introduce SIZE, and take this opportunity to review the relevant notions from the theory of the simply-typed lambda calculus.

- In Chapter 3 we then introduce Vec, a programming language based on Size and supporting a vector-type constructor.
- In Chapter 4 we then define shape analysis as a translation from Vec to Size and prove some of its properties.
- In Chapter 5 we introduce a different kind of shape analysis, again as a translation from Vec to (a variant of) Size.
- In Chapter 6 we investigate ways of extending shape analysis to other language constructs and features, such as data conditionals and polymorphism.
- Finally, in Chapter 7 we summarize the work and draw conclusions.

The systems and techniques described in this thesis have been implemented in Standard ML and the implementation is freely available. The details on downloading and descriptions of the systems can be found in Appendix B.

Chapter 2

The Size language

This chapter introduces Size, a functional language based on the simply typed lambda calculus with finite products and the type of natural numbers. As we shall see in the following chapters, Size will be the target language for the shape analysis of a vector-based language, Vec, and it is constructed with this purpose in mind. The chapter consists of three sections. In the first section we review the relevant aspects of typed lambda calculi, mainly in order to standardise notation used later in the thesis. The second section discusses some extensions of the core calculus, those that will be used in the definitions of Size and, in the next chapter, of Vec. Finally, the third section introduces the Size language itself. The material in the first two sections is quite standard, and most of it can be found in any reference on lambda calculus, such as [Barendregt, 1984] and [Hindley and Seldin, 1986] for the untyped version, and [Lambek and Scott, 1986], [Girard et al., 1989] or [Barendregt, 1992] for typed systems.

2.1 Typed Lambda Calculi

The main idea behind lambda calculus is that of function abstraction and application. If a function f is defined by some expression t whose value depends on the argument x

$$f : x \mapsto t[x] \ ,$$

then we can abstract over x to get the lambda term $\lambda x.t$ denoting the function f. When we want to apply f to an argument t', notation $f\ t'$, we have to substitute t' for x into the expression t. This process is reflected in the basic equational rule of lambda calculus, β equality:

$$(\lambda x.t)\ t' = t[t'/x]\ .$$

Starting from this tenet, we can build a formal calculus of functions, the (untyped) lambda calculus. This calculus is at the foundations of all functional languages.

2.1.1 Types and terms

Typed lambda calculus was first introduced in [Curry, 1934] and [Church, 1940] as a restriction of the untyped version where only terms for which a valid type can be inferred are considered legal. The two approaches, Curry's and Church's, though equivalent in their expressive power, have different flavours: the terms of a lambda calculus *à la* Curry are the terms of the untyped lambda calculus, and each such term has an associated set of possible types (which may be empty), while the terms of a lambda calculus *à la* Church are annotated with types and one can derive the type of the term from the annotations within the term. The rest of this chapter will concentrate on lambda calculi *à la* Church, but all of the material can be presented in Curry's style as well.

Definition 2.1.1 The *types* of the simply typed lambda calculus (over a set *Base* of base types) are given by the following grammar

$$\sigma\ ::=\ \delta \mid \sigma \to \sigma$$

where $\delta \in \mathit{Base}$ is a base type. Types will be denoted by lowercase Greek letters.

Intuitively, the type $\sigma_1 \to \sigma_2$ is the type of functions from σ_1 to σ_2 (sometimes called the *exponential* type and denoted $\sigma_2^{\sigma_1}$). The operator $\to$ is right-associative, so the expression $\sigma_1 \to \sigma_2 \to \sigma_3$ is parsed as $\sigma_1 \to (\sigma_2 \to \sigma_3)$.

As we mentioned above, terms presented in Church's style should be annotated with their types. However, it turns out that it is only necessary to annotate those variables that are being abstracted over – the types of the remaining term constructs can then be inferred, and are unique (in a given typing context, see below). This is reflected in the syntax of the terms.

In the following we shall suppose that we are given a (countably large) set *Var* of variables. The terms of the calculus are defined as follows.

Definition 2.1.2 The set of *raw terms* t is given by the following grammar

$$t \ ::= \ x^\sigma \mid \lambda x^\sigma.t \mid t\ t$$

where is a type, and $x \in \mathit{Var}$ a variable.

Remarks

1. We shall be using letters from the end of the alphabet, such as x, y, and z to denote arbitrary variables; f and g to denote variables of (typically) function types; t, u, and v to denote arbitrary terms.

2. In the term x^σ σ is the *type* of the variable x. The type superscript may be omitted when appropriate.

3. The term $\lambda x^\sigma.t$ is called an *abstraction*. As above, σ is the type of x and may be omitted. The *scope* of λx^σ in $\lambda x^\sigma.t$ is t. We may sometimes omit successive λ's and write $\lambda x, y.t$ instead of $\lambda x.\lambda y.t$.

4. The term $t_1\ t_2$ is an *application*. Application associates to the left, i.e. $t_1\ t_2\ t_3$ is parsed as $(t_1\ t_2)\ t_3$.

5. In following proofs, we shall often use induction on *the complexity rank* of a term t. This is defined as follows – the complexity rank of a variable is one, the complexity rank of an abstraction $\lambda x.t$ being one plus the complexity of t and the complexity rank of an application being one plus the sum of the ranks of the two subterms. We may also refer to this induction as the induction on *the structural complexity* of t.

6. Symbol $\equiv$ will denote the *syntactic* equality, both on types and terms.

Definition 2.1.3 A *typing context* Γ is an assignment $\{x_1 : \sigma_1, \ldots, x_n : \sigma_n\}$ of types to a finite set of variables (with no variable x_i occurring twice). We put $Dom(\Gamma) = \{x_1, \ldots, x_n\}$. If Γ is the context $\{x_1 : \sigma_1, \ldots, x_n : \sigma_n\}$, then $\Gamma, x : \sigma$ denotes the context $\{x_1 : \sigma_1, \ldots, x_n : \sigma_n, x : \sigma\}$ (provided $x \notin Dom(\Gamma)$). We say that in a context Γ a raw term t *has a type* σ if the judgment $\Gamma \vdash t : \sigma$ is derivable from the rules given in Fig 2.1.

Variable	$x : \sigma \vdash x^\sigma : \sigma$
Weakening	$\dfrac{\Gamma \vdash t : \sigma \quad x \notin Dom(\Gamma)}{\Gamma, x : \sigma' \vdash t : \sigma}$
Abs	$\dfrac{\Gamma, x : \sigma \vdash t : \sigma'}{\Gamma \vdash \lambda x^\sigma . t : \sigma \to \sigma'}$
App	$\dfrac{\Gamma \vdash t : \sigma' \to \sigma \quad \Gamma \vdash t' : \sigma'}{\Gamma \vdash t\ t' : \sigma}$

Figure 2.1: Typing judgments for the simply typed lambda calculus

Remarks

1. When $\Gamma = \{\}$ is the empty typing context, we may write $\vdash t : \sigma$ instead of $\Gamma \vdash t : \sigma$.

2. The derivation of $\Gamma \vdash t : \sigma$ can be viewed as a tree. Similarly, derivations in other formal systems introduced later in this thesis can be looked at as trees. Since various characteristics of such trees can (and will) be used as induction ranks, it might be useful to clarify the terminology. The *height* of a tree is the length of its longest branch. The *complexity* of a tree is one plus the sum of the complexities of all its immediate subtrees, with the complexity of a leaf being one.

Definition 2.1.4 A raw term t is *legal* if there exists a typing context Γ and a type σ such that

$$\Gamma \vdash t : \sigma \ .$$

The following shows that a raw term has at most one type in a given typing context, thus justifying our decision to restrict type annotations to variables in abstractions.

Lemma 2.1.5 *Let t be a raw term. If $\Gamma \vdash t : \sigma_0$ and $\Gamma \vdash t : \sigma_1$, then $\sigma_0 \equiv \sigma_1$.*

Proof. The proofs of most of the lemmas in this section are simple inductive proofs, and we shall omit most of them. We prove this lemma for illustration. The proof proceeds by induction on the structure of t.

1. If $t \equiv x$ then both $x : \sigma_0 \in \Gamma$ and $x : \sigma_1 \in \Gamma$. But since a variable has at most one type in a given typing context, $\sigma_0 \equiv \sigma_1$.

2. If $t \equiv \lambda x^\tau.t'$ then the last step in the derivation of $\Gamma \vdash t : \sigma_0$ had to be

$$\frac{\Gamma, x : \tau \vdash t' : \sigma_0'}{\Gamma \vdash \lambda x^\tau.t' : \tau \to \sigma_0'}$$

   where $\sigma_0 \equiv \tau \to \sigma_0'$ for some type σ_0'. Similarly $\sigma_1 \equiv \tau \to \sigma_1'$ where $\Gamma, x : \tau \vdash t' : \sigma_1'$. We now apply induction hypothesis to t' to get $\sigma_0' \equiv \sigma_1'$ and therefore also $\sigma_0 \equiv \sigma_1$.

3. If $t \equiv t_1\ t_2$ then $\Gamma \vdash t_1 : \tau_0 \to \sigma_0$ for some τ_0, and $\Gamma \vdash t_1 : \tau_1 \to \sigma_1$ for some τ_1. By applying the induction hypothesis to t_1 we get $\tau_0 \to \sigma_0 \equiv \tau_1 \to \sigma_1$ and thus $\sigma_0 \equiv \sigma_1$.

∎

2.1.2 Manipulating terms

We now define the sets of occurrences and subterms of a term (for a formal definition of occurrences see e.g. [Dershowitz and Jouannaud, 1990]). We need the following lemma to ensure that Definition 2.1.7 makes sense.

Lemma 2.1.6 A subterm of a legal term is legal.

Proof. By induction on the structure of the term. ∎

Definition 2.1.7 The set of *occurrences* of a (raw) term t is the set $O(t)$ of sequences of natural numbers defined inductively as follows

- If t is a variable, then $O(t) = \{e\}$ where e is the empty sequence.
- If $t \equiv \lambda x.t'$ is an abstraction, then $O(t) = \{e\} \cup \{1 \cdot s,\ s \in O(t')\}$ where $\cdot$ represents concatenation.
- If $t \equiv t_1\ t_2$ is an application, then $O(t) = \{e\} \cup \{i \cdot s,\ s \in O(t_i),\ i \in \{1, 2\}\}$.

The *subterm of* t *at occurrence* s, denoted $t|_s$, is, for $s \in O(t)$, given inductively by

$$\begin{array}{rcl} t|_e & = & t \\ \lambda x.t'|_{1\cdot s} & = & t'|_s \\ (t_1\ t_2)|_{i\cdot s} & = & t_i|_s, \quad i \in \{1,2\} \end{array}$$

and it is undefined otherwise.

If $t|_s = t'$, we also say that s *indexes* t' *in* t.

The operation of subterm replacement is defined below.

Definition 2.1.8 The operation of *replacing a subterm of* t *at an occurrence* s *by a term* u (notation $t[u]_s$) is defined for $s \in O(t)$ by

$$\begin{array}{rcl} t[u]_e & = & u \\ \lambda x.t'[u]_{1\cdot s} & = & \lambda x.t'[u]_s \\ (t_1\ t_2)[u]_{1\cdot s} & = & t_1[u]_s\ t_2 \\ (t_1\ t_2)[u]_{2\cdot s} & = & t_1\ t_2[u]_s \end{array}$$

and it is undefined otherwise.

The previous definition leads to the definition of a context (see, for example, [Barendregt, 1984]), a structure we will occasionally need:

Definition 2.1.9 Let σ be a type. The set of *raw contexts C with holes of type σ* is given by the following grammar

$$C \ ::= \ x \mid \Box^\sigma \mid \lambda x^\tau.C \mid C\ C$$

where x is a variable and τ a type.

We can extend the type inference mechanism for raw terms to raw contexts in the obvious way, with the type of $\Box^\sigma$ being σ. A *context* is then a raw context for which a type can be inferred. Contexts are thus, in effect, legal terms containing occurrences of the symbol $\Box^\sigma$ (called simply "the hole").

Remarks

1. We shall use capital roman letters, mostly C, to denote contexts. We may sometimes write $C[\,]^\sigma$ to indicate that C is a context with holes of type σ.

2. Note that a context as defined above may contain more than one hole – this is the way contexts are defined in, for example, [Barendregt, 1984]. Other authors, such as [Klop, 1992], prefer contexts with a single hole only.

3. Since there will not be any chance of confusion, we shall extend $\equiv$ to denote the syntactic equality of contexts. Also, we define the notion of an occurrence of a subcontext in a (raw) context, analogously to the operation on (raw) terms.

We will use $C[t]$ to denote the term resulting from replacing all occurrences of $\Box$ in a context C by a term t (using the operation of subterm replacement defined above).

Lemma 2.1.10 Let $C[\,]^\sigma$ be a context, t a term, and let $\Gamma \vdash C : \tau$ and $\Gamma \vdash t : \sigma$. Then

$$\Gamma \vdash C[t] : \tau\ .$$

Proof. By induction on the height of the derivation of $\Gamma \vdash C : \tau$. ∎

Definition 2.1.11 An occurrence s indexing a variable x in a term t is *free* if there is no prefix s' of s indexing a term of the form $\lambda x^\sigma.t'$. The occurrence s is *bound* otherwise. A variable is *free* in a term if it occurs free in it. A term with no free variables is *closed*.

In other words, an occurrence of a variable x is bound if it is in the scope of some λx^σ. The set of free variables of t is denoted $FV(t)$. Similarly we define the set of free variables of a context C.

Note that the operation of hole replacement defined on contexts can *capture* free variables, that is, if x occurs free in t, then its corresponding occurrence in $C[t]$ may be bound.

The following lemma describes the relationship between free variables in a term and valid typing contexts.

Lemma 2.1.12 *If* $\Gamma \vdash t : \sigma$*, then* $FV(t) \subseteq Dom(\Gamma)$*. Moreover, there exists a context* Γ' *such that* $Dom(\Gamma') = FV(t)$ *and* $\Gamma' \vdash t : \sigma$*.*

Proof. By induction on the height of the judgment derivation. ∎

In particular, if t is closed and has a type σ under some typing context Γ, then $\vdash t : \sigma$. It therefore makes sense to talk about the type of a closed term without mentioning any typing context.

Before we can introduce substitution, we have to address the complications arising from α-equivalence. If we think of a lambda abstraction as defining a function, then the name of the formal parameter, i.e. the variable used in the abstraction, is irrelevant. We thus want to work with terms "up to the renaming of bound variables", or, in other words, "up to α-equivalence". Two approaches can be taken: either we introduce α conversion as a special rewrite rule on the set of terms, or we can identify α-equivalent terms at the syntactic level. We will opt for the latter approach, and the formal treatment is given below.

Definition 2.1.13 The α-equivalence relation $\equiv_\alpha$ is the congruence on the set of raw terms generated by the rule

$$\lambda x^\sigma.t \equiv_\alpha \lambda y^\sigma.t'$$

where $y \notin FV(t)$ and t' is the term resulting from replacing every free occurrence of x in t by y.

Lemma 2.1.14 shows that a term α-equivalent to a legal term is legal.

Lemma 2.1.14 If $\Gamma \vdash t : \sigma$ and $t \equiv_\alpha t'$, then $\Gamma \vdash t' : \sigma$.

Proof. By induction on the judgment derivation. ∎

The previous lemma justifies the following definition.

Definition 2.1.15 An *untyped term* is an equivalence class under $\equiv_\alpha$. A *term* is an untyped term consisting of legal terms.

In the following, we shall write $t \equiv t'$ to indicate that t and t' represent the same term (or the same untyped term), i.e. they are in the same equivalence class under $\equiv_\alpha$. It is useful to note that $\equiv_\alpha$ is decidable.

Even though terms are now whole equivalence classes, we shall typically work with individual representatives of these classes. The formal approach requires us to prove that definitions and theorems stated in terms of these representatives do not depend on how they were chosen. However, since such proofs are typically straightforward, we shall omit them.

Definition 2.1.16 If t and t' are untyped terms, and x a variable, then the result of *substituting t' for x in t*, denoted $t[t'/x]$, is given inductively by

- $x[t'/x] \equiv t'$.
- $y[t'/x] \equiv y$ where $x \not\equiv y$.
- $(\lambda x.t_1)[t'/x] \equiv \lambda x.t_1$.
- $(\lambda y^\sigma.t_1)[t'/x] \equiv \lambda y^\sigma.t_1[t'/x]$ where $x \not\equiv y$ and $y \notin FV(t')$.
- $(\lambda y^\sigma.t_1)[t'/x] \equiv \lambda z^\sigma.(t_1[z/y])[t'/x]$ where $x \not\equiv y$, $y \in FV(t')$ and $z \notin FV(t_1) \cup FV(t')$.
- $(t_1\ t_2)[t'/x] \equiv t_1[t'/x]\ t_2[t'/x]$.

Note that the definition above does not make sense unless we work with terms up to α-equivalence, since, in the fifth clause above, the variable z is not determined by the rule and can be any fresh variable.

The following lemma shows that substituting legal terms for variables of the proper type into legal terms results again in legal terms.

Lemma 2.1.17 If $\Gamma, x : \sigma' \vdash t : \sigma$ and $\Gamma \vdash t' : \sigma'$, then $\Gamma \vdash t[t'/x] : \sigma$.

Proof. By induction on the derivation of $\Gamma, x : \sigma' \vdash t : \sigma$. ∎

Lemma 2.1.18 (Substitution lemma). *Let t, t' and t'' be untyped terms, x and y variables, such that $x \not\equiv y$ and $x \notin FV(t'')$. Then*

$$t[t'/x][t''/y] \equiv t[t''/y][t'[t''/y]/x] \ .$$

Proof. By induction on the structure of t. ∎

The previous lemma shows that, under certain conditions, the order of substitutions is irrelevant. This ensures that the following definition makes sense.

Definition 2.1.19 Let $x_1, \ldots, x_n$ be a list of variables with no variable occurring twice, and $t_1, \ldots, t_n$ a list of closed terms. A *substitution* $S = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ is a mapping from untyped terms to untyped terms given by

$$S(t) \equiv t[t_1/x_1] \ldots [t_n/x_n] \ .$$

Let S be a substitution and y a variable. $S[y \mapsto t']$ will then denote the substitution

$$\{x_1 \mapsto t_1, \ldots, x_{i-1} \mapsto t_{i-1}, x_{i+1} \mapsto t_{i+1}, x_n \mapsto t_n, y \mapsto t'\}$$

if $y \equiv x_i$ for some i, and the substitution $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n, y \mapsto t'\}$ otherwise.

The definition of substitution given above is not the most general possible, but it will be sufficient for our needs.

Later in the chapter, we will have to "substitute" contexts for free variables in other contexts. Substitution, as defined in Definition 2.1.16, is an operation defined only on terms, not on contexts. Moreover, we cannot simply extend it to contexts, since its definition relies on working with terms up to α-equivalence, and we do not identify α-equivalent contexts. We therefore introduce the following operation of *variable replacement*

Definition 2.1.20 Let C be a context, C' a closed context and x a variable. The context C *with the free occurrences of x replaced by C'*, notation $C\{C'/x\}$, is defined inductively as follows.

$$\begin{array}{rcl} x\{C'/x\} & \equiv & C' \\ y\{C'/x\} & \equiv & y \quad \text{when } y \not\equiv x \\ \Box\{C'/x\} & \equiv & \Box \\ (\lambda x.C_1)\{C'/x\} & \equiv & \lambda x.C_1 \\ (\lambda y.C_1)\{C'/x\} & \equiv & \lambda y.(C_1\{C'/x\}) \quad \text{when } y \not\equiv x \\ (C_1\ C_2)\{C'/x\} & \equiv & C_1\{C'/x\}\ C_2\{C'/x\} \end{array}$$

The definition may look complicated, but its intuitive meaning is simple – to get the context $C\{C'/x\}$, one has to replace every free occurrence of x in C by C'. As C' is closed, the question of variable capture does not arise.

2.1.3 Terms and equality

Definition 2.1.21 The β equational theory of the simply typed lambda calculus is the congruence $=_\beta$ on the set of terms generated by the schema

$$(\lambda x.t)\ t' =_\beta t[t'/x]\ .$$

As we mentioned before, the rule schema above is called the β equality.

We will sometimes be interested in the decidability of various equational theories, and for this it is often useful to regard the equations as having a computational aspect, e.g. to view the β equality as a rewrite rule

$$(\lambda x.t)\ t' \to_\beta t[t'/x]$$

describing the *computation* of the result of a function application. Before we go any further, let us define some standard notions from the theory of term rewriting systems (for a more detailed account of various aspects of rewriting see, for example, [Klop, 1992] or [Dershowitz and Jouannaud, 1990]).

Definition 2.1.22 A set of *base reductions* is simply a binary relation R on the set of terms. When $(t_1, t_2) \in R$ then we say that t_1 is an *R-redex* and t_2 an *R-reduct*. The relation $\to_R$, called the *one-step R-reduction*, is defined as follows

$$\frac{(t_1, t_2) \in R}{t_1 \to_R t_2}$$

$$\frac{t_1 \to_R t_2}{\lambda x.t_1 \to_R \lambda x.t_2}$$

$$\frac{t_1 \to_R t_2}{t'\ t_1 \to_R t'\ t_2}$$

$$\frac{t_1 \to_R t_2}{t_1\ t' \to_R t_2\ t'}$$

where t' is a term of the proper type. The *many-step R-reduction* $\to_R^*$ is the reflexive and transitive closure of $\to_R$. *R-equality* $=_R$ is the symmetric and transitive closure of $\to_R^*$.

Remarks

1. The one-step R-reduction can alternatively be defined by

$$\frac{(t_1, t_2) \in R}{C[t_1] \to_R C[t_2]}$$

   where C is a context with one hole of the proper type.

2. Note that $=_\beta$ as defined above is indeed the congruence generated by the base reduction β.

Definition 2.1.23 A relation $\to_R$ is *confluent* (or *Church-Rosser*) if, for any pair of reductions $t \to_R^* t'$ and $t \to_R^* t''$, there is a term s such that $t' \to_R^* s$ and $t'' \to_R^* s$. Relation $\to_R$ is *locally confluent* (or *weakly Church-Rosser*) if whenever $t \to_R t'$ and $t \to_R t''$, there is a term s such that $t' \to_R^* s$ and $t'' \to_R^* s$.

Definition 2.1.24 A term t is a *normal form under* $\to_R$, if there is no term t' such that $t \to_R t'$. If $t' \to_R^* t$ for some normal form t, we say that t is a *normal form of* t'. Relation $\to_R$ is *weakly normalising* if every term has a normal form. Relation $\to_R$ is *strongly normalising* if every reduction sequence $t_1 \to_R t_2 \to_R t_3 \to_R \cdots$ terminates.

If a reduction is both confluent and strongly normalising, then the equality $=_R$ is decidable (provided, of course, that R is computable, but this we will always assume). From this point of view the following lemma is useful.

Lemma 2.1.25 (Newman's lemma). *A reduction which is strongly normalising and locally confluent is confluent.*

Proof. A short proof can be found in [Dershowitz and Jouannaud, 1990]. ∎

Lemma 2.1.26 (Subject reduction lemma). If $\Gamma \vdash t : \tau$ and $t \to_\beta^* t'$, then $\Gamma \vdash t' : \tau$.

Proof. It is sufficient to show the lemma for one step β-reductions as the general case then immediately follows. So let $t \to_\beta t'$. This is induced by some base reduction $s \to_\beta s'$. Lemma 2.1.17 ensures that the type of s is the same as the type of s' (under any valid typing context). The lemma that easily follows by structural induction on t. ∎

Theorem 2.1.27 *$\to_\beta$ is both strongly normalising and confluent.*

Proof. A proof of strong normalisation can be found in [Girard et al., 1989]. The proof of confluency is a direct corollary of the previous lemma and the confluency of the untyped lambda calculus, the proof of which can be found in [Barendregt, 1984]. ∎

Corollary 2.1.28 *$=_\beta$ is decidable.*

Proof. Follows immediately from the previous theorem. We first normalise both terms in question (any reduction strategy will do), and then compare the resulting normal forms (the decidability of the last step follows from the decidability of α-equivalence). ∎

When we extend the β reduction system by the rule

$$\lambda x.t\ x \to t$$

(called the η rule), we get another widely studied system – the $\beta\eta$ equality. This stronger kind of term equality captures the notion of extensional equality – two closed terms t and t' of type σ are *extensionally equal* if, for any context $C[\ ]^{\sigma}$ of base type, $C[t]$ and $C[t']$ have the same normal form. Although the $\beta\eta$ system is still confluent and strongly normalising, the confluency is easily lost once we extend the base calculus by new constructs and their corresponding η rules (such as products, see Section 2.2.2). There are ways to restore the confluency of the calculus, such as to view η rules as *expansions* rather than contractions, as in [Jay and Ghani, 1995] or [Ghani, 1995], but this leads to non-terminating reductions, and therefore syntactic restrictions on the use of this rule have to be imposed. Since extensionality of the calculus will not be our main concern, η rules will not be introduced in any of the languages in this thesis.

2.1.4 Evaluating terms

If we now go back to the motivating example at the beginning of this section, we can see that we can really think of terms as representing (very simple) computations and the process of a reduction to a β normal form as an evaluation of this computation. However, even though a reduction system tells us what the result of this computation should be, it does not specify how to obtain this result. Of course, in a confluent and strongly normalising system this is not much of a problem, since any strategy will ultimately lead to the same result. But even in such a system, some strategies may lead to the result much faster than others. And when working with either non-confluent or non-strongly normalising systems, the choice of a strategy becomes even more important.

The two standard evaluation strategies (or *operational semantics*) are *eager* (call-by-value) and *lazy* (call-by-name). Informally, an eager operational semantics will evaluate the argument to a function before β reducing the application, while a lazy one will reduce the β-redex straight away. The remainder of this section will formally present an eager operational semantics for the

simply typed lambda calculus. This semantics will be later extended to Size and Vec. A good introduction to operational semantics is [Gunter, 1992].

Definition 2.1.29 A term t is a *value* if it is either a variable or of the form $\lambda x^\sigma.t'$.

Definition 2.1.30 The *eager operational semantics* of the simply typed lambda calculus is the binary relation $\Rightarrow$ between closed terms of the calculus generated by the rules in Fig 2.2.

$$\lambda x^\sigma.t \Rightarrow \lambda x^\sigma.t$$

$$\frac{t_1 \Rightarrow \lambda x^\sigma.t \quad t_2 \Rightarrow v \quad t[v/x] \Rightarrow v'}{t_1\ t_2 \Rightarrow v'}$$

Figure 2.2: Eager operational semantics of the simply typed calculus

Definition 2.1.31 Given a term t, we say that the *evaluation of t terminates*, if there exists a value v such that $t \Rightarrow v$ (in which case t *evaluates to* v). The evaluation *does not terminate* otherwise.

Remark The complexity of the derivation tree of $t \Rightarrow v$ will be referred to as the *evaluation complexity* of t.

The following lemma summarises the important facts about operational semantics.

Lemma 2.1.32 *Let t and v be closed terms.*

1. *If $t \Rightarrow v$, then v is a value.*
2. *If $t \Rightarrow v_1$ and $t \Rightarrow v_2$, then $v_1 \equiv v_2$.*
3. *If $t \Rightarrow v$, then $t \rightarrow_\beta^* v$.*
4. *If $\vdash t : \sigma$ and $t \Rightarrow v$, then $\vdash v : \sigma$.*

5. *The evaluation of t terminates.*

Proof.

1. By induction on the height of the derivation of $t \Rightarrow v$.
2. By induction on the height of the derivation of $t \Rightarrow v$ and then by case analysis on the last rule used.
3. By induction on the height of the derivation of $t \Rightarrow v$.
4. Follows immediately from the previous clause and Lemma 2.1.26.
5. By induction on the structure of t.

∎

One might think that the third clause in the previous lemma can be strengthened to say that whenever $t \Rightarrow v$, then v is the β-normal form of t. This is not necessarily true, however, since a value can contain β-redexes hidden under λ's.

Definition 2.1.33 Let t and t' be closed terms of type σ. We say that t and t' have *equivalent values* (notation $t \cong t'$), if either both evaluate to the same value or the evaluation of both does not terminate.

Definition 2.1.34 Let t and t' be closed terms of type σ. We say that t *operationally approximates* t' (notation $t \preceq t'$) if, for any closed context $C[\ ]^\sigma$ of base type,

$$C[t] \Rightarrow v \quad \text{implies} \quad C[t'] \Rightarrow v \ .$$

Lemma 2.1.35 $\preceq$ is reflexive and transitive.

Proof. Both properties follow immediately from the definition. ∎

Lemma 2.1.35 justifies the following definition.

Definition 2.1.36 Let t and t' be closed terms of type σ. If $t \preceq t'$ and $t' \preceq t$, then we say that t and t' are *operationally equivalent* (notation $t \approx t'$).

It is easy to see that $t \approx t'$ iff $C[t] \cong C[t']$ for every C of the proper form.

2.2 Extending the calculus

The base lambda calculus, as presented in the previous section, is not, of course, very expressive. There are, essentially, two ways of increasing its expressive power – either by enriching its type system by new constructs or by extending its term formation rules (or, often, both). In this section we describe several such extensions, those that are relevant to the definitions of Size and Vec. Most of them are quite standard and have been used in various functional languages, such as PCF (see e.g. [Gunter, 1992]).

2.2.1 Combinators

The type system of the simply typed lambda calculus is built over some set *Base* of base types. However, it is easy to see that there are no closed terms of these base types. One has to introduce some combinators (or additional term constructors) to be able to construct such terms. Usually it is a matter of taste and personal preference whether to use combinators or constructors, and often the expressive powers of both approaches are, in effect, equivalent. The real difference lies in the operational semantics, since combinators are forced to use the "standard" evaluation strategy, while constructors (such as the conditional below) can use specially tailored evaluation mechanisms. We will, in most cases, prefer the use of combinators over constructors. So in the following we shall suppose that we have a set of combinators *Comb* together with a function *ctype* which assigns to each combinator its type. The set of raw terms is then defined by

$$t \ ::= \ x \mid c \mid \lambda x^{\sigma}.t \mid t\ t$$

where $c \in Comb$. The typing judgments are extended by the inference rule

$$\textbf{Combinator} \quad \frac{ctype(c) \equiv \sigma}{\Gamma \vdash c : \sigma}$$

(instead of $ctype(c) \equiv \sigma$, we will usually write $c : \sigma$). We will probably want to extend $\to_\beta$ by some other rules describing the computation of terms involving combinators (such rules are usually called δ rules). Of course, if we do that, it is not clear whether the resulting calculus will still be strongly normalising and/or confluent.

2.2.2 Product type

One of the most common extensions of the type system is the introduction of product types (so much so, that often, as in [Girard et al., 1989] or [Lambek and Scott, 1986], the calculus without products is not studied at all).

We extend the type system by the *product* type constructor $\times$ with $\sigma \times \sigma'$ representing the binary product of σ and σ'. $\times$ binds tighter than $\to$ and associates to the right. We introduce three combinators

$$\begin{array}{lcl} \mathsf{pair}^{\sigma,\sigma'} & : & \sigma \to \sigma' \to \sigma \times \sigma' \\ \mathsf{fst}^{\sigma,\sigma'} & : & \sigma \times \sigma' \to \sigma \\ \mathsf{snd}^{\sigma,\sigma'} & : & \sigma \times \sigma' \to \sigma' \end{array}$$

representing the pairing, and the first and second projection, respectively. Where the types are clear from the context or unimportant, we will omit them. We introduce the notation $\langle t, t' \rangle$ as syntactic sugar for $\mathsf{pair}\ t\ t'$.

We can think of pair as a *constructor*, and fst and snd as *destructors*. That the combinators should come in such pairs is not surprising – they are the equivalents of pairs of introduction and elimination rules in logic.

The usual reduction rules associated with products are

$$\begin{array}{lcl} \mathsf{fst}\ \langle t, t' \rangle & \to & t \\ \mathsf{snd}\ \langle t, t' \rangle & \to & t' \end{array}$$

Another commonly used rule is the η rule which for products has usually the form

$$\langle \mathsf{fst}\ t, \mathsf{snd}\ t \rangle \ \to\ t\ ,$$

but, as we said before, we do not introduce this rule.

2.2.3 Unit

The unit type un is the trivial type with only one closed normal term un, also called the unit. The rule most frequently associated with the unit is the η contraction

$$t \ \rightarrow \ \mathsf{un}$$

but, again, we do not introduce this rule.

2.2.4 Natural numbers

The type nat of natural numbers is a very common base type. There are several possible ways to build terms of this type, and we will opt for a very simple one. We introduce three combinators

$$\begin{array}{lcl} \mathsf{zero} & : & \mathsf{nat} \\ \mathsf{succ} & : & \mathsf{nat} \rightarrow \mathsf{nat} \\ \mathsf{pred} & : & \mathsf{nat} \rightarrow \mathsf{nat} \end{array}$$

representing zero, and the successor and predecessor functions, respectively. The associated reduction rule is

$$\mathsf{pred}\ (\mathsf{succ}\ t) \ \rightarrow \ t\ .$$

In some situations, one could also add the rule

$$\mathsf{pred}\ \mathsf{zero} \ \rightarrow \ \mathsf{zero}\ .$$

However, we prefer to regard $\mathsf{pred}\ \mathsf{zero}$ as an error, and leave it, for now, to be an irreducible term.

One can also add combinators representing other arithmetic operations, such as addition or multiplication, and the appropriate reduction rules, but, for the moment, we restrict ourselves to the combinators given above.

We introduce the following notation to improve readability

$$\tilde{\mathsf{n}} \equiv \underbrace{\mathsf{succ}\ (\cdots(\mathsf{succ}}_{\mathsf{n}\text{-times}}\ \mathsf{zero}))\ .$$

We will use lowercase roman letters, typically n and m, to denote terms of type nat.

2.2.5 Booleans and conditionals

The base type bool is a type with two closed normal terms, true and false. Had we introduced the coproduct type constructor $+$, booleans could have been represented by the type $\mathsf{un} + \mathsf{un}$ with true and false having been given by the first and second injection, respectively.

We extend the term-forming rules by a new constructor, the conditional

$$\mathsf{if}\ t\ \mathsf{then}\ t\ \mathsf{else}\ t\ .$$

In a term $\mathsf{if}\ t\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2$, t is the *condition*, t_1 is the *then branch*, and t_2 is the *else branch*. The typing of conditionals is given by

$$\textbf{Conditional}\quad \frac{\Gamma \vdash t : \mathsf{bool} \quad \Gamma \vdash t_1 : \sigma \quad \Gamma \vdash t_2 : \sigma}{\Gamma \vdash \mathsf{if}\ t\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2 : \sigma}$$

and its base reductions are

$$\begin{array}{lcl} \mathsf{if\ true\ then}\ t_1\ \mathsf{else}\ t_2 & \to & t_1 \\ \mathsf{if\ false\ then}\ t_1\ \mathsf{else}\ t_2 & \to & t_2\ . \end{array}$$

The main reason for introducing conditional as a term constructor, rather than a combinator, is because its evaluation should be lazy, i.e. only the branch determined by the value of the condition should be evaluated. This becomes especially important in the presence of recursion (see below).

Of course, the conditional constructor described above is only one, though the most common, of many possible ways how to introduce branching into a programming language. In the SIZE language we will introduce a slightly modified conditional (denoted ifs) which branches according to a natural number, with $\tilde{\mathsf{0}}$ indicating one branch and successors the other. The reduction rules are then

$$\begin{array}{lcl} \mathsf{ifs}\ \tilde{\mathsf{0}}\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2 & \to & t_1 \\ \mathsf{ifs\ succ}\ t\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2 & \to & t_2\ . \end{array}$$

We will also need "boolean" operations on sizes, such as conjunction and negation given by

$$\begin{array}{lcl} \mathsf{and} & : & \mathsf{nat} \to \mathsf{nat} \to \mathsf{nat} \\ \mathsf{and} & \equiv & \lambda x.\lambda y.\mathsf{ifs}\ x\ \mathsf{then}\ y\ \mathsf{else}\ x \\ \mathsf{not} & : & \mathsf{nat} \to \mathsf{nat} \\ \mathsf{not} & \equiv & \lambda x.\mathsf{ifs}\ x\ \mathsf{then\ succ}\ x\ \mathsf{else}\ x\ . \end{array}$$

2.2.6 Equality

Equality is probably the most common predicate. If we want to introduce equality over type σ as a combinator eq^σ, one of its possible types, and the one we prefer, is

$$\mathsf{eq}^\sigma \;:\; \sigma \to \sigma \to \mathsf{bool}\ .$$

The associated reduction rules then depend on the type σ. In the simplest case, when $\sigma \equiv \mathsf{un}$, the reduction rule is

$$\mathsf{eq}^{\mathsf{un}}\ \mathsf{un}\ \mathsf{un} \quad\to\quad \mathsf{true}$$

More interesting and useful is equality on natural numbers, where the reductions can be

$$\begin{array}{lcl} \mathsf{eq}^{\mathsf{nat}}\ \mathsf{zero}\ \mathsf{zero} & \to & \mathsf{true} \\ \mathsf{eq}^{\mathsf{nat}}\ (\mathsf{succ}\ t_1)\ (\mathsf{succ}\ t_2) & \to & \mathsf{eq}^{\mathsf{nat}}\ t_1\ t_2 \\ \mathsf{eq}^{\mathsf{nat}}\ (\mathsf{succ}\ t_1)\ \mathsf{zero} & \to & \mathsf{false} \\ \mathsf{eq}^{\mathsf{nat}}\ \mathsf{zero}\ (\mathsf{succ}\ t_2) & \to & \mathsf{false} \end{array}$$

We might easily extend equality to products of natural numbers, where two pairs are equal if both components are (note that this requires introducing the conjunction operation $\mathsf{and} : \mathsf{bool} \to \mathsf{bool} \to \mathsf{bool}$ on booleans):

$$\mathsf{eq}^{\mathsf{nat}\times\mathsf{nat}}\ \langle t_1, t_2\rangle\ \langle t_1', t_2'\rangle \quad\to\quad \mathsf{and}\ (\mathsf{eq}^{\mathsf{nat}}\ t_1\ t_1')\ (\mathsf{eq}^{\mathsf{nat}}\ t_2\ t_2')$$

Equality over function types is usually of a limited use only, since the kind of equality one would typically like to capture (the extensional one) is then often undecidable. We will not introduce equality over function types in any of our languages.

2.2.7 Recursion

The untyped lambda calculus is a very powerful system, and it can be shown that any recursive function can be represented in it. But the type restrictions we imposed in the simply typed lambda calculus have severely reduced its expressive power. To recover this expressivity, we have to be able to represent unbounded recursion in some way. We opt for introducing the rec constructor:

$$\mathsf{rec}\ f^\sigma.t$$

where f is variable. The typing of rec is given below.

$$\textbf{Recursion} \quad \frac{\Gamma, f : \sigma \vdash t : \sigma}{\Gamma \vdash \mathsf{rec}\ f^{\sigma}.t : \sigma}$$

(type superscripts will often be omitted). As we can see, the variable f is bound in rec $f.t$. This means that one has to be careful when defining substitution into such a term. The relevant clauses extending the definition of substitution (Definition 2.1.16) look as follows

$$\begin{array}{lcl}(\mathsf{rec}\ x.t)[t'/x] & \equiv & \mathsf{rec}\ x.t \\ (\mathsf{rec}\ f.t)[t'/x] & \equiv & \mathsf{rec}\ f.t[t'/x] \quad \text{where } x \not\equiv f \text{ and } f \notin FV(t') \\ (\mathsf{rec}\ f.t)[t'/x] & \equiv & \mathsf{rec}\ z.(t[z/y])[t'/x] \\ & & \text{where } x \not\equiv y,\ f \in FV(t') \text{ and } z \notin FV(t) \cup FV(t')\end{array}$$

The reduction rule associated with recursion is

$$\mathsf{rec}\ f.t \quad \rightarrow \quad t[\mathsf{rec}\ f.t/f]$$

Of course, this rule destroys any hope for strong normalisation of the system, since, when f is free in t, we can keep applying this rule forever. Nevertheless, the evaluation of rec $f.t$ may terminate even in these cases, due to the lazy evaluation of conditionals. The rec constructor is powerful enough to represent, when taken together with the other constructs defined above (conditionals, natural numbers), any recursive function.

One could also introduce the less powerful *iterator* constructor iter

$$\mathsf{iter} : (\sigma \rightarrow \sigma) \rightarrow \sigma \rightarrow \mathsf{nat} \rightarrow \sigma$$

with the reduction rules

$$\begin{array}{lcl}\mathsf{iter}\ f\ t\ (\mathsf{succ}\ n) & \rightarrow & f(\mathsf{iter}\ f\ t\ n) \\ \mathsf{iter}\ f\ t\ \mathsf{zero} & \rightarrow & t\end{array}$$

Unlike the unbounded recursion, iteration always terminates, and can be used to represent primitive recursive functions only.

2.2.8 Errors

There are several ways of treating terms which are usually regarded as errors (such as taking the predecessor of zero, or division by zero). One can take the view that terms containing errors do not have to evaluate to the "correct" value, and introduce some ad hoc reduction rules for such terms (such as $\mathsf{pred\ zero} \to \mathsf{zero}$). Another possibility is to leave such terms unreduced, and let the evaluation mechanism to deal with them (e.g. through the use of exceptions). A common method (mainly in untyped calculi), is to regard as errors those terms whose evaluation does not terminate. This, of course, is not possible if one wants to have a strongly normalising system. Since the detection of errors is our primary objective, we will make errors explicit in the language (actually only certain kinds of errors – shape errors). We will introduce, for each type σ, the *error* term, denoted err^σ (the type may be omitted where appropriate). We then put, for example,

$$\mathsf{pred\ zero} \quad \to \quad \mathsf{err}^{\mathsf{nat}} \ .$$

A question then arises whether terms involving errors should, at least in some cases, themselves be considered to be errors. This could be reflected in the reduction system by introducing error propagating rules such as

$$f \ \mathsf{err} \to \mathsf{err}$$

(and analogous ones for other term constructs). Though regarding $f\ \mathsf{err}$ as error may look reasonable, introduction of the rule above (and its equivalents) leads to a loss of confluency in the calculus. This can be easily seen by considering, for example, the term $(\lambda x.y)\ \mathsf{err}$ which can be reduced either to y (by the β-reduction), or to err. We will follow the ideas from [Plotkin, 1975] and [Moggi, 1989] and restrict β reduction to *values*, i.e. terms that cannot evaluate to error. We will not introduce rules propagating errors as yet, however, and let the operational semantics handle them. A different approach to error handling will be investigated in Chapter 5.

2.3 The SIZE language

We can now define the SIZE language. As we have already said, it is a simply typed functional language with finite products, the type of natural numbers,

and general recursion. The decisions taken in its design were motivated by its intended use as a target language of shape analysis of array-based languages.

SIZE type system is given by

$$\sigma ::= \mathsf{un} \mid \mathsf{sz} \mid \sigma \times \sigma \mid \sigma \to \sigma \ .$$

The *base* types are un and sz, the *ground* types θ are built up from the base ones using the product construction only.

The type sz, called *size*, has exactly the same term structure as the type of natural numbers as presented in the previous section. The different name should emphasize the fact that sz will be used as the type of sizes of arrays, as opposed to, for example, the type of data entries.

The raw terms are given by the following grammar

$$t \ ::= \ x \mid c \mid \lambda x^\sigma.t \mid t\ t \mid \mathsf{rec}\ f^\sigma.t \mid \mathsf{ifs}\ t\ \mathsf{then}\ t\ \mathsf{else}\ t\ .$$

where c is a combinator (the set of which is given in Fig 2.3, together with their types). All the combinators were already discussed in the previous section, though, in some cases, the types have been slightly modified. Note, in particular, that we do not introduce equality of functions. Also, as we have already indicated in Section 2.2.5, we will use sizes as conditions in shape conditionals, and the following terms as logical operations on sizes:

$$\begin{array}{lcl} \mathsf{and} & : & \mathsf{sz} \to \mathsf{sz} \to \mathsf{sz} \\ \mathsf{and} & \equiv & \lambda x.\lambda y.\mathsf{ifs}\ x\ \mathsf{then}\ y\ \mathsf{else}\ x \\ \mathsf{not} & : & \mathsf{sz} \to \mathsf{sz} \\ \mathsf{not} & \equiv & \lambda x.\mathsf{ifs}\ x\ \mathsf{then}\ \mathsf{succ}\ x\ \mathsf{else}\ x\ . \end{array}$$

The type inference rules are given in Fig 2.4.

SIZE values are defined below.

Definition 2.3.1 A *value* v is a term given by the following grammar

$$v \ ::= \ x \mid c\ (\not\equiv \mathsf{err}) \mid \lambda x^\sigma.t \mid \mathsf{pair}\ v \mid \ \mathsf{pair}\ v\ v \mid \mathsf{succ}\ v \mid \mathsf{eq}\ v \ \ .$$

A *result* is a term which is either a value or err.

Remarks

un	:	un
$\mathsf{pair}^{\sigma,\sigma'}$	:	$\sigma \to \sigma' \to \sigma \times \sigma'$
$\mathsf{fst}^{\sigma,\sigma'}$	:	$\sigma \times \sigma' \to \sigma$
$\mathsf{snd}^{\sigma,\sigma'}$	:	$\sigma \times \sigma' \to \sigma'$
zero	:	sz
succ	:	$\mathsf{sz} \to \mathsf{sz}$
pred	:	$\mathsf{sz} \to \mathsf{sz}$
eq^{θ}	:	$\theta \to \theta \to \mathsf{sz}$
err^{σ}	:	σ

Figure 2.3: Size combinators and their types

Variable $$x : \sigma \vdash x : \sigma$$

Weakening $$\frac{\Gamma \vdash t : \sigma \quad x \notin Dom(\Gamma)}{\Gamma, x : \sigma' \vdash t : \sigma}$$

Combinator $$\frac{c : \sigma}{\Gamma \vdash c : \sigma}$$

Abs $$\frac{\Gamma, x : \sigma \vdash t : \sigma'}{\Gamma \vdash \lambda x^{\sigma}.t : \sigma \to \sigma'}$$

App $$\frac{\Gamma \vdash t : \sigma' \to \sigma \quad \Gamma \vdash t' : \sigma'}{\Gamma \vdash t\ t' : \sigma}$$

Rec $$\frac{\Gamma, f : \sigma \vdash t : \sigma}{\Gamma \vdash \mathsf{rec}\ f^{\sigma}.t : \sigma}$$

Ifs $$\frac{\Gamma \vdash t : \mathsf{sz} \quad \Gamma \vdash t_1 : \sigma \quad \Gamma \vdash t_2 : \sigma}{\Gamma \vdash \mathsf{ifs}\ t\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2 : \sigma}$$

Figure 2.4: Typing judgments for Size

1. We will use the letters v and r to denote values and results, respectively.
2. Values are built from constructors (such as pair) and partially applied destructors (eq).
3. The evaluation of a term in the operational semantics of SIZE will either not terminate or return a result (see below).

Lemma 2.3.2 *If* $\Gamma \vdash t : \sigma_0$ *and* $\Gamma \vdash t : \sigma_1$ *then* $\sigma_0 \equiv \sigma_1$.

Proof. By induction on the height of derivation of $\Gamma \vdash t : \sigma_0$. ∎

The base reductions are given in Fig 2.5. Notice the modified β (and some other) rules that ensure eager reduction (along the lines of [Plotkin, 1975]). As we have discussed in Section 2.2.8, additional rules propagating err could be added. This would simplify normal forms of the system, but as we will not concentrate on normal forms here, we do not introduce them. We will come back to this problem in Chapter 5.

$(\lambda x.t)\ v$	$\rightarrow_{\mathsf{sz}}$	$t[v/x]$
$\mathsf{rec}\ f.t$	$\rightarrow_{\mathsf{sz}}$	$t[\mathsf{rec}\ f.t/f]$
$\mathsf{fst}\ \langle t_1, v\rangle$	$\rightarrow_{\mathsf{sz}}$	t_1
$\mathsf{snd}\ \langle v, t_2\rangle$	$\rightarrow_{\mathsf{sz}}$	t_2
$\mathsf{pred\ zero}$	$\rightarrow_{\mathsf{sz}}$	err
$\mathsf{pred}\ (\mathsf{succ}\ t)$	$\rightarrow_{\mathsf{sz}}$	t
$\mathsf{eq\ zero}\ t$	$\rightarrow_{\mathsf{sz}}$	t
$\mathsf{eq}\ (\mathsf{succ}\ t)\ \mathsf{zero}$	$\rightarrow_{\mathsf{sz}}$	$\mathsf{succ}\ t$
$\mathsf{eq}\ (\mathsf{succ}\ t_1)\ (\mathsf{succ}\ t_2)$	$\rightarrow_{\mathsf{sz}}$	$\mathsf{eq}\ t_1\ t_2$
$\mathsf{eq\ un\ un}$	$\rightarrow_{\mathsf{sz}}$	zero
$\mathsf{eq}\ \langle t_1, t_2\rangle\ \langle s_1, s_2\rangle$	$\rightarrow_{\mathsf{sz}}$	$\mathsf{and}\ (\mathsf{eq}\ t_1\ s_1)\ (\mathsf{eq}\ t_2\ s_2)$
$\mathsf{ifs\ zero\ then}\ t_1\ \mathsf{else}\ t_2$	$\rightarrow_{\mathsf{sz}}$	t_1
$\mathsf{ifs\ succ}\ v\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2$	$\rightarrow_{\mathsf{sz}}$	t_2

Figure 2.5: Base reductions for SIZE

The usual suite of basic lemmas, such as that presented earlier in the chapter for the core calculus, can then be proved. We will concentrate here only on those claims whose proofs require significant changes.

Theorem 2.3.3 $\rightarrow_{\mathsf{sz}}$ *is confluent.*

Proof. The detailed proof of this theorem is given in Appendix A, and we shall therefore present here only its brief overview. The proof relies on the confluency of any *left-linear* and *non-ambiguous* combinatory reduction system, as showed in [Klop, 1980]. Here left-linear has its usual meaning, that is in every reduction rule a variable occurs at most once in the redex (i.e. on the left), and non-ambiguous is the equivalent of having no critical pairs. Such reduction systems are *regular* and we can construct, in a routine way, a regular combinatory reduction system Ω such that SIZE can be embedded into Ω in such a way that the reduction $\rightarrow_{\mathsf{sz}}$ is preserved. The confluency of $\rightarrow_{\mathsf{sz}}$ then immediately follows. ∎

Of course, $\rightarrow_{\mathsf{sz}}$ is not strongly normalising, because of the presence of general recursion.

We now present an operational semantics of SIZE.

Lemma 2.3.4 Let v be a value.

- If $v : \mathsf{sz}$, then there exists n such that $v \equiv \tilde{\mathsf{n}}$.
- If $v : \sigma_1 \times \sigma_2$, then $v \equiv \langle v_1, v_2 \rangle$ for some values $v_1 : \sigma_1$ and $v_2 : \sigma_2$.

Proof. Follows immediately from the definition. ∎

The operational semantics $\Rightarrow_{\mathsf{sz}}$ of SIZE is the binary relation between closed terms generated by the rules in Fig 2.6. The semantics is an extension of the eager semantics for the simply typed lambda calculus. The rules are structured so as to avoid ambiguity in evaluation as much as possible.

Lemma 2.3.5 *If* $t \Rightarrow_{\mathsf{sz}} r$*, then* r *is a result.*

Proof. By induction on the height of the derivation of $t \Rightarrow_{\mathsf{sz}} r$. ∎

$$\frac{r \text{ is a result}}{r \Rightarrow_{\mathsf{sz}} r} \qquad \frac{t_1 \Rightarrow_{\mathsf{sz}} \lambda x.t \quad t_2 \Rightarrow_{\mathsf{sz}} v \quad t[v/x] \Rightarrow_{\mathsf{sz}} r}{t_1\ t_2 \Rightarrow_{\mathsf{sz}} r}$$

$$\frac{t_1 \Rightarrow_{\mathsf{sz}} \mathsf{err}}{t_1\ t_2 \Rightarrow_{\mathsf{sz}} \mathsf{err}} \qquad \frac{t_1 \Rightarrow_{\mathsf{sz}} v \quad t_2 \Rightarrow_{\mathsf{sz}} \mathsf{err}}{t_1\ t_2 \Rightarrow_{\mathsf{sz}} \mathsf{err}}$$

$$\frac{t_1 \Rightarrow_{\mathsf{sz}} \mathsf{pair} \quad t_2 \Rightarrow_{\mathsf{sz}} v}{t_1\ t_2 \Rightarrow_{\mathsf{sz}} \mathsf{pair}\ v} \qquad \frac{t_1 \Rightarrow_{\mathsf{sz}} \mathsf{pair}\ v_1 \quad t_2 \Rightarrow_{\mathsf{sz}} v_2}{t_1\ t_2 \Rightarrow_{\mathsf{sz}} \langle v_1, v_2\rangle}$$

$$\frac{t_1 \Rightarrow_{\mathsf{sz}} \mathsf{fst} \quad t_2 \Rightarrow_{\mathsf{sz}} \langle v_1, v_2\rangle}{t_1\ t_2 \Rightarrow_{\mathsf{sz}} v_1} \qquad \frac{t_1 \Rightarrow_{\mathsf{sz}} \mathsf{snd} \quad t_2 \Rightarrow_{\mathsf{sz}} \langle v_1, v_2\rangle}{t_1\ t_2 \Rightarrow_{\mathsf{sz}} v_2}$$

$$\frac{t_1 \Rightarrow_{\mathsf{sz}} \mathsf{succ} \quad t_2 \Rightarrow_{\mathsf{sz}} v}{t_1\ t_2 \Rightarrow_{\mathsf{sz}} \mathsf{succ}\ v} \qquad \frac{t_1 \Rightarrow_{\mathsf{sz}} \mathsf{pred} \quad t_2 \Rightarrow_{\mathsf{sz}} \mathsf{succ}\ v}{t_1\ t_2 \Rightarrow_{\mathsf{sz}} v}$$

$$\frac{t_1 \Rightarrow_{\mathsf{sz}} \mathsf{pred} \quad t_2 \Rightarrow_{\mathsf{sz}} \mathsf{zero}}{t_1\ t_2 \Rightarrow_{\mathsf{sz}} \mathsf{err}} \qquad \frac{t_1 \Rightarrow_{\mathsf{sz}} \mathsf{eq} \quad t_2 \Rightarrow_{\mathsf{sz}} v}{t_1\ t_2 \Rightarrow_{\mathsf{sz}} \mathsf{eq}\ v}$$

$$\frac{t_1 \Rightarrow_{\mathsf{sz}} \mathsf{eq}\ \mathsf{zero} \quad t_2 \Rightarrow_{\mathsf{sz}} v}{t_1\ t_2 \Rightarrow_{\mathsf{sz}} v} \qquad \frac{t_1 \Rightarrow_{\mathsf{sz}} \mathsf{eq}\ (\mathsf{succ}\ v) \quad t_2 \Rightarrow_{\mathsf{sz}} \mathsf{zero}}{t_1\ t_2 \Rightarrow_{\mathsf{sz}} \mathsf{succ}\ v}$$

$$\frac{t_1 \Rightarrow_{\mathsf{sz}} \mathsf{eq}\ (\mathsf{succ}\ v) \qquad t_2 \Rightarrow_{\mathsf{sz}} \mathsf{succ}\ v' \qquad \mathsf{eq}\ v\ v' \Rightarrow_{\mathsf{sz}} v''}{t_1\ t_2 \Rightarrow_{\mathsf{sz}} v''} \qquad \frac{t_1 \Rightarrow_{\mathsf{sz}} \mathsf{eq}\ \mathsf{un} \quad t_2 \Rightarrow_{\mathsf{sz}} \mathsf{un}}{t_1\ t_2 \Rightarrow_{\mathsf{sz}} \mathsf{zero}}$$

$$\frac{t_1 \Rightarrow_{\mathsf{sz}} \mathsf{eq}\ \langle v_1, v_2\rangle \qquad t_2 \Rightarrow_{\mathsf{sz}} \langle v_1', v_2'\rangle \qquad \mathsf{and}\ (\mathsf{eq}\ v_1\ v_1')\ (\mathsf{eq}\ v_2\ v_2') \Rightarrow_{\mathsf{sz}} v}{t_1\ t_2 \Rightarrow_{\mathsf{sz}} v} \qquad \frac{t[\mathsf{rec}\ f.t/f] \Rightarrow_{\mathsf{sz}} r}{\mathsf{rec}\ f.t \Rightarrow_{\mathsf{sz}} r}$$

$$\frac{t \Rightarrow_{\mathsf{sz}} \mathsf{zero} \quad t_1 \Rightarrow_{\mathsf{sz}} r}{\mathsf{ifs}\ t\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2 \Rightarrow_{\mathsf{sz}} r} \qquad \frac{t \Rightarrow_{\mathsf{sz}} \mathsf{succ}\ v \quad t_2 \Rightarrow_{\mathsf{sz}} r}{\mathsf{ifs}\ t\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2 \Rightarrow_{\mathsf{sz}} r}$$

$$\frac{t \Rightarrow_{\mathsf{sz}} \mathsf{err}}{\mathsf{ifs}\ t\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2 \Rightarrow_{\mathsf{sz}} \mathsf{err}}$$

Figure 2.6: Operational semantics of SIZE

Unlike in the simply typed lambda calculus, there are terms in SIZE, such as $\mathsf{rec}\ f^{\sigma}.f$, whose evaluation does not terminate.

Lemma 2.3.6 *If $t \Rightarrow_{\mathsf{sz}} r_1$ and $t \Rightarrow_{\mathsf{sz}} r_2$, then $r_1 \equiv r_2$.*

Proof. By induction on the height of the derivation of $t \Rightarrow_{\mathsf{sz}} r_1$ and then by case analysis on the last rule used. A little care has to be taken when evaluating terms such as succ zero, as then there are two possible derivations: either succ zero evaluates to itself in one step, since it is a value, or we can treat it as an application and first evaluate both succ and zero to themselves. Of course, the result is the same in both cases. ∎

If a term evaluates to a value, then its evaluation respects the $\rightarrow_{\mathsf{sz}}$ reduction, as the following lemma shows.

Lemma 2.3.7 *If $t \Rightarrow_{\mathsf{sz}} v$ (where $v \not\equiv \mathsf{err}$), then $t \rightarrow^{*}_{\mathsf{sz}} v$.*

Proof. By induction on the height of the derivation of $t \Rightarrow_{\mathsf{sz}} v$. ∎

Lemma 2.3.8 *If $\vdash t : \sigma$ and $t \Rightarrow_{\mathsf{sz}} r$, then $\vdash r : \sigma$.*

Proof. Follows immediately from the Subject reduction lemma and the previous lemma. ∎

In the remainder of this chapter, we will aim to prove the Soundness theorem for SIZE (Theorem 2.3.14), which says that reduction preserves operational equality:

$$t =_{\mathsf{sz}} t' \qquad \text{implies} \qquad t \approx_{\mathsf{sz}} t'$$

The proof of this theorem will require several technical results, which we will prove first. These technical parts may be skipped without losing understanding of the rest of the thesis.

It might be useful at this point to review some of the notation introduced earlier:

$t \equiv t'$	means	t and t' are syntactically equal, see Definition 2.1.2
$t \Rightarrow r$		t evaluates to r, see Definition 2.1.31
$t \cong t'$		t and t' have equivalent results, see Definition 2.1.33
$t \approx t'$		t and t' are operationally equivalent, see Definition 2.1.36
$t = t'$		t and t' are equal (under the equational theory), see Definition 2.1.21

(We will omit the subscripts as in $\Rightarrow_{\mathsf{sz}}$ in the remainder of this chapter as all of the above notions will refer to SIZE.)

Remember that operational equivalence of two terms t and t' essentially means "$C[t]$ and $C[t']$ have the same value for any context C of base type" (contexts were defined in Definition 2.1.9).

Lemma 2.3.9 Let $t : \sigma$ be a closed term, $C[\ ]^\sigma$ a closed context.

1. If $t \Rightarrow r$ and $C[t]$ is a result, then so is $C[r]$.
2. If $C[t]$ is a result, then either t is a result or each occurrence of $\square$ in C is in the scope of a λ. Consequently, if $C[t]$ is a result of ground type, then either t is a result, or there are no occurrences of $\square$ in C.
3. If $C[t]$ and $C[t']$ are results and at least one of t and t' is not a result then every occurrence of $\square$ in C is in the scope of a λ.

Proof. The second clause can be proved by induction on the structure of C. The first and third clause immediately follow from the second. ∎

Lemma 2.3.10 Let t and t' be closed terms of type σ with the same result r, and $C[\ ]^\sigma$ a context such that the evaluation of $C[t]$ terminates. Then there exists a context $\widehat{C}[\ ]^\sigma$ such that $C[t] \Rightarrow \widehat{C}[t]$ and $C[t'] \Rightarrow \widehat{C}[t']$. In particular, the evaluation of $C[t']$ terminates.

Proof.

Let $C[t] \Rightarrow r'$. If $t \equiv r \equiv t'$ for some result r, then we can put $\widehat{C} \equiv r'$ and the lemma immediately holds. So in the following we will assume that either $t \not\equiv r$ or $t' \not\equiv r$.

The proof proceeds by induction on the evaluation complexity of $C[t]$, and then by case analysis on the structure of C. Note that we will use in this proof (as well as in some other later on) typing restrictions to eliminate those cases that are ill typed.

1. If $C \equiv \square$, then let $\widehat{C} \equiv r$.

2. If C is a combinator c, then let $\widehat{C} \equiv c$.

3. If $C \equiv \lambda x.C_1$, then let $\widehat{C} \equiv C$.

4. Suppose $C \equiv C_1\ C_2$.

   We can apply the induction hypothesis to $C_1[t]$ to get a context $\widehat{C}_1$ such that $C_1[t] \Rightarrow \widehat{C}_1[t]$ and $C_1[t'] \Rightarrow \widehat{C}_1[t']$.

   Suppose $\widehat{C}_1[t] \equiv \mathsf{err}$. Then $\widehat{C}_1$ cannot be $\Box$ by Lemma 2.3.9 (3) and thus $\widehat{C}_1 \equiv \mathsf{err}$. Now let $\widehat{C} \equiv \mathsf{err}$ and the lemma holds.

   So assume that $\widehat{C}_1[t]$ is not err. Then the evaluation complexity of $C_2[t]$ is less than that of $C[t]$ and we can use the induction hypothesis on $C_2[t]$ to get a context $\widehat{C}_2$ such that $C_2[t] \Rightarrow \widehat{C}_2[t]$ and $C_2[t'] \Rightarrow \widehat{C}_2[t']$. If $\widehat{C}_2[t] \equiv \mathsf{err}$ then, as above, $\widehat{C}_2 \equiv \mathsf{err}$. Now let $\widehat{C} \equiv \mathsf{err}$ and the lemma holds. So in the following assume that $\widehat{C}_2[t]$ is not err.

   If both $\widehat{C}_1[t]\ \widehat{C}_2[t]$ and $\widehat{C}_1[t']\ \widehat{C}_2[t']$ are values, we put $\widehat{C} \equiv \widehat{C}_1\ \widehat{C}_2$ and the proof is finished. So suppose $\widehat{C}_1[t]\ \widehat{C}_2[t]$ is not a value (the case when $\widehat{C}_1[t']\ \widehat{C}_2[t']$ is not a value is analogous). Then $\widehat{C}_1[t]$ has to be of one of the following forms.

   (a) If $\widehat{C}_1[t] \equiv \lambda x.t_1$ then, by Lemma 2.3.9 (3), $\widehat{C}_1 \equiv \lambda x.C'$ for some C'. Consider the context $C'' \equiv C'\{\widehat{C}_2/x\}$ (see Definition 2.1.20). Clearly $C''[t] \cong C[t]$ and $C''[t'] \cong C[t']$. Also, the evaluation complexity of $C''[t]$ is less than that of $C[t]$ and we can therefore apply the induction hypothesis to get context $\widehat{C}''$ such that $C''[t] \Rightarrow \widehat{C}''[t]$ and $C''[t'] \Rightarrow \widehat{C}''[t']$. Let then $\widehat{C} \equiv \widehat{C}''$ and the lemma holds.

   (b) If $\widehat{C}_1[t] \equiv \mathsf{fst}$ then $\widehat{C}_2[t] \equiv \langle u_1, u_2\rangle$ for some values u_1 and u_2. Lemma 2.3.9 (3) then implies that $\widehat{C}_1 \equiv \mathsf{fst}$ and $\widehat{C}_2 \equiv \langle C_2', C_2''\rangle$. Since the evaluation complexity of $C_2'[t]$ is less than that of $C[t]$, we use the induction hypothesis to get $\widehat{C}_2'$ such that $C_2'[t] \Rightarrow \widehat{C}_2'[t]$ and $C_2'[t'] \Rightarrow \widehat{C}_2'[t']$. Let then $\widehat{C} \equiv \widehat{C}_2'$. Analogously for $\widehat{C}_1[t] \equiv \mathsf{snd}$.

   (c) In the remaining cases, when $\widehat{C}_1[t] \in \{\mathsf{pred}, \mathsf{eq}\ u'\}$, the context $\widehat{C}_2[t]$ is of ground type and therefore, by Lemma 2.3.9 (3), there are no occurrences of $\Box$ in $\widehat{C}_1\ \widehat{C}_2$. We then put $\widehat{C} \equiv v'$ (where $\widehat{C}_1[t]\ \widehat{C}_2[t] \Rightarrow v'$) and the result follows.

5. Suppose $C \equiv \mathsf{rec}\ f.C_1$. Consider the context $C_2 \equiv C_1\{\mathsf{rec}\ f.C_1/f\}$. The evaluation complexity of $C_2[t]$ is less than that of $C[t]$ and we can therefore apply the induction hypothesis to get context $\widehat{C}_2$ such that $C_2[t] \Rightarrow \widehat{C}_2[t]$ and $C_2[t'] \Rightarrow \widehat{C}_2[t']$. We then put $\widehat{C} \equiv \widehat{C}_2$.

6. If $C \equiv \mathsf{ifs}\ C_1\ \mathsf{then}\ C_2\ \mathsf{else}\ C_3$ then we can apply the induction hypothesis to $C_1[t]$ to get $\widehat{C}_1$. There are three cases to consider.

   (a) If $\widehat{C}_1[t] \equiv \mathsf{err}$ then, by Lemma 2.3.9 (3), $\widehat{C}_1 \equiv \mathsf{err}$ and we put $\widehat{C} \equiv \mathsf{err}$.

   (b) If $\widehat{C}_1[t] \equiv \mathsf{zero}$ then, by Lemma 2.3.9 (3), $\widehat{C}_1 \equiv \mathsf{zero}$ and we can apply the induction hypothesis to $C_2[t]$ to get $\widehat{C}_2$. We then put $\widehat{C} \equiv \widehat{C}_2$.

   (c) The case when $\widehat{C}_1[t] \equiv \mathsf{succ}\ v$ is analogous to the previous one.

∎

We now introduce the notion of a strict context and strict operational equivalence (sometimes called applicative bisimulation). This will simplify following proofs.

Definition 2.3.11 Consider the following grammar:

$$G \ ::= \Box \mid G\ v \mid \mathsf{fst}\ G \mid \mathsf{snd}\ G\ .$$

where v is a value. Contexts $G[\,]$ of base type generated by this grammar are *strict*.

Two closed terms t and t' of type σ are *strictly operationally equivalent* (notation $t \sim t'$) if $G[t] \cong G[t']$ for any strict context $G[\,]^\sigma$.

We shall use the letter G to represent strict contexts. The following lemma shows that the notion of strict operational equivalence coincides with the standard operational equivalence:

Lemma 2.3.12 Let t and t' be closed terms of type σ. Then

$$t \sim t' \quad \text{iff} \quad t \approx t'\ .$$

Proof. The "if" direction follows immediately. For the converse, suppose $t \sim t'$. We have to show that $C[t] \cong C[t']$ for every context $C[\,]^\sigma$ of base type. We shall write $C[\,]$ in the form $G[C'[\,]]$ where $G[\,]$ is strict and a maximal such. The proof proceeds by induction on the evaluation complexity of $G[C'[t]]$, then by induction on the structural complexity of $C'[\,]$ and finally by case analysis on the structure of $C'[\,]$.

1. If $C' \equiv \Box$ then $C[t] \equiv G[t] \cong G[t'] \equiv C[t']$.
2. If C' is a combinator c then $C[t] \equiv C[t']$.
3. If $C' \equiv \lambda x.C_1$ then $G \equiv G'[\Box\ v]$ for some strict $G'[\,]$ and value v and we get

$$\begin{array}{rcll} C[t] & \equiv & G'[(\lambda x.C_1[t])\ v] & \\ & \cong & G'[C_1[t][v/x]] & \\ & \equiv & G'[C_1[v/x][t]] & \\ & \cong & G'[C_1[v/x][t']] & \text{by induction on evaluation complexity} \\ & \cong & G'[(\lambda x.C_1[t'])\ v] \equiv C[t'] & \end{array}$$

4. Suppose $C' \equiv C_1\ C_2$. Then, by twice applying Lemma 2.3.10, we get contexts $\widehat{C}_1[\,]$ and $\widehat{C}_2[\,]$ such that $C_1[t] \Rightarrow \widehat{C}_1[t]$, $C_1[t'] \Rightarrow \widehat{C}_1[t']$, $C_2[t] \Rightarrow \widehat{C}_2[t]$ and $C_2[t'] \Rightarrow \widehat{C}_2[t']$. Then either $\widehat{C}_1 \equiv \Box$ and thus $t \equiv t'$ and the proof is finished, or one of the following cases arises

   (a) If $\widehat{C}_1 \equiv \lambda x.D$ then

$$\begin{array}{rcll} C[t] & \cong & G[(\lambda x.D[t])\ \widehat{C}_2[t]] & \\ & \cong & G[D[t][\widehat{C}_2[t]/x]] & \\ & \equiv & G[D\{\widehat{C}_2/x\}[t]] & \text{(see Definition 2.1.20 for } D\{\widehat{C}_2/x\}) \\ & \cong & G[D\{\widehat{C}_2/x\}[t']] & \text{by induction on eval. complexity} \\ & \cong & G'[(\lambda x.D[t'])\ \widehat{C}_2[t']] \cong C[t'] & \end{array}$$

   (b) The remaining cases when $\widehat{C}_1 \equiv \{\mathsf{pair}, \mathsf{pair}\ \ D, \mathsf{fst}, \mathsf{snd}, \ldots\}$ are straightforward.

5. Suppose $C' \equiv \mathsf{rec}\ f.C_1$.

$$\begin{array}{rcll} C[t] & \equiv & G[\mathsf{rec}\ f.C_1[t]] & \\ & \cong & G[C_1[t][\mathsf{rec}\ f.C_1[t]/x]] & \\ & \equiv & G[C_1\{\mathsf{rec}\ f.C_1/x\}[t]] & \text{(see Definition 2.1.20)} \\ & \cong & G[C_1\{\mathsf{rec}\ f.C_1/x\}[t']] & \text{by induction on eval. complexity} \\ & \cong & G'[\mathsf{rec}\ f.C_1[t']] \cong C[t'] & \end{array}$$

6. Suppose $C' \equiv \mathsf{ifs}\ C_1\ \mathsf{then}\ C_2\ \mathsf{else}\ C_3$. Then, by applying Lemma 2.3.10, we get $\widehat{C}_1[\ ]$ such that $C_1[t] \Rightarrow \widehat{C}_1[t]$ and $C_1[t'] \Rightarrow \widehat{C}_1[t']$. There are three cases to consider.

   (a) If $\widehat{C}_1[t] \equiv \mathsf{err}$ then either $\widehat{C}_1 \equiv \mathsf{err}$ and thus $C[t] \cong \mathsf{err} \cong C[t']$, or $\widehat{C}_1 \equiv \Box$ and $t \equiv \mathsf{err} \equiv t'$ and the proof is finished.

   (b) If $\widehat{C}_1[t] \equiv \mathsf{zero}$ then either $\widehat{C}_1 \equiv \mathsf{zero}$ or $\widehat{C}_1 \equiv \Box$ and $t \equiv \mathsf{zero} \equiv t'$. In either case

   $$C[t] \cong G[C_2[t]] \cong G[C_2[t']] \cong C[t]$$

   by induction on the structure of $C'[\ ]$.

   (c) The case $\widehat{C}_1[t] \equiv \mathsf{succ}\ v$ is analogous to the previous one.

∎

The following lemma shows that base reductions on Size respect operational equivalence.

Lemma 2.3.13 Let $t \rightarrow_{\mathsf{sz}} t'$ be a base reduction. Then for any substitution S of values for the free variables of t, $S(t) \cong S(t')$.

Proof. It is easy to check the lemma for each base reduction, and we will therefore show only the most complicated case – the β reduction.

Suppose $t \equiv (\lambda x.s)\ v$ (and thus $t' \equiv s[v/x]$). Then, for any substitution S as specified above, $S(v)$ is a value (and, in particular, $v \not\equiv \mathsf{err}$) and thus

$$S(t) \equiv S((\lambda x.s)\ v) \equiv \lambda x.S(s)\ S(v) \cong S(s[S(v)/x]) \equiv S(t')$$

and the proof is finished. ∎

We can now prove the Soundness theorem for SIZE:

Theorem 2.3.14 (Soundness theorem). *Let s be a closed term. If $s =_{\mathsf{sz}} s'$, then $s \approx_{\mathsf{sz}} s'$.*

Proof. It is enough to show that for any two closed terms s and s', whenever $s \rightarrow^*_{\mathsf{sz}} s'$ then $s \approx s'$. It is sufficient to consider the one-step reductions only as the general case then immediately follows. Since any one-step reduction is induced by a base reduction, this means that $s \equiv C[t]$, $s' \equiv C[t']$ for some context $C[\,]$ and base reduction $t \rightarrow_{\mathsf{sz}} t'$.

So let $t \rightarrow_{\mathsf{sz}} t'$ be a base reduction. It is sufficient to show that for any context $C[\,]$ and any substitution S of values such that $S(C[t])$ is closed,

$$S(C[t]) \approx S(C[t']) \ .$$

The proof proceeds by induction on the structural complexity of $C[\,]$ and then by case analysis on the structure of $C[\,]$. The only two non-trivial cases follow from the two previous lemmas, as shown below.

1. If $C \equiv \Box$ then $S(C[t]) \equiv S[t] \cong S[t'] \equiv S(C[t'])$ by Lemma 2.3.13.

2. If $C \equiv \lambda x^{\sigma'}.C'$ then, by Lemma 2.3.12, it is enough to show that

   $$S(C[t])\ v \approx S(C[t])\ v$$

   for any value of type σ'. We now get

   $$\begin{array}{rcl} S(C[t])\ v & \cong & S(C'[t])[v/x] \\ & \equiv & S'(C'[t]) \quad \text{where } S' = S[x \mapsto v] \\ & \equiv & S'(C'[t']) \quad \text{by induction on the structure of } C \\ & \equiv & S(C'[t'])[v/x] \\ & \cong & S(C[t']) \end{array}$$

∎

The Soundness theorem has several important corollaries that are stated below:

Corollary 2.3.15

1. *If* $t \Rightarrow r$*, then* $t \approx r$.
2. *Let* t *and* t' *be closed terms of type* σ *which evaluate to the same result. Then* $t \approx t'$.

Proof. The first clause follows easily from Theorem 2.3.14 and the second is an immediate corollary of the first. ∎

Chapter 3

The VEC language

We now extend the SIZE language defined in the previous chapter by the vector type constructor, vec, and call the resulting language VEC. Since the vector construction can be nested, one can represent arrays of arbitrary (finite) dimensions in VEC. The recursion mechanism of SIZE, incorporated to VEC, is powerful enough to express most common array-based operations, particularly those of linear algebra. Its expressivity combined with its simplicity makes VEC an ideal language for study of shape analysis, and it will be used to this purpose in the following chapter. Note that efficiency was not the primary consideration in the design of VEC, rather it is just a vehicle for demonstrating the validity of our methods. As we discuss in Chapter 7, the ideas of VEC have already been incorporated into a "proper" programming language, FISh ([Jay and Steckler, 1998]).

The chapter is structured as follows. In the first section we discuss ways of adding vectors and arrays to a functional language. The second section then defines VEC itself, and in the last section we use it to represent various array-based operations.

3.1 Vectors and Arrays

Introducing vectors (or, more often, lists) to a functional language is normally straightforward. Typically, one adds a new type constructor, vec, and a suite

of suitable combinators, such as

$$\begin{array}{lcl} \mathsf{nil}^\tau & : & \mathsf{vec}\ \tau \\ \mathsf{cons}^\tau & : & \tau \to \mathsf{vec}\ \tau \to \mathsf{vec}\ \tau \\ \mathsf{hd}^\tau & : & \mathsf{vec}\ \tau \to \tau \\ \mathsf{tl}^\tau & : & \mathsf{vec}\ \tau \to \mathsf{vec}\ \tau \\ \mathsf{length}^\tau & : & \mathsf{vec}\ \tau \to \mathsf{nat} \end{array}$$

(where vec binds tighter then $\to$). Here nil^τ represents the empty vector (of terms of type τ), $\mathsf{cons}^\tau\ t_1\ t$ prepends a term t_1 to a vector t, hd^τ returns the first entry in a vector (its *head*), and tl^τ the remaining part of the vector (its *tail*). As usual, type superscripts will be usually omitted.

The usual reduction rules associated with the above combinators are

$$\begin{array}{lcl} \mathsf{hd}\ (\mathsf{cons}\ t_1\ t) & \to & t_1 \\ \mathsf{tl}\ (\mathsf{cons}\ t_1\ t) & \to & t\ . \\ \mathsf{length}\ (\mathsf{nil}) & \to & \mathsf{zero} \\ \mathsf{length}\ (\mathsf{cons}\ t_1\ t) & \to & \mathsf{succ}\ (\mathsf{length}\ t)\ . \end{array}$$

Neither the head nor the tail of an empty vector is usually defined.

In the next section a similar approach will be used to add vectors to VEC, a functional language based on SIZE. To make shape analysis of VEC possible, we have to make two changes. Firstly, we modify the type of length to

$$\mathsf{length}^\tau\ :\ \mathsf{vec}\ \tau \to \mathsf{sz}$$

since the length of an array is a shape (a size) rather than a datum (a natural number). Secondly, for technical reasons which will become clear in the following chapter, we do not allow empty vectors in VEC – the simplest vector which can be constructed will be the singleton. Therefore, instead of the nil constructor, we introduce the singleton combinator

$$\mathsf{sing}^\tau\ :\ \tau \to (\mathsf{vec}\ \tau)$$

with the reduction rules

$$\begin{array}{lcl} \mathsf{hd}\ (\mathsf{sing}\ t) & \to & t \\ \mathsf{tl}\ (\mathsf{sing}\ t) & \to & \mathsf{err} \\ \mathsf{length}\ (\mathsf{sing}\ t) & \to & \tilde{}1\ . \end{array}$$

We introduce the following notation to improve readability:

$$\begin{array}{lcl} t_1 :: t & \equiv & \mathsf{cons}\ t_1\ t \\ [t_1, t_2, \ldots, t_n] & \equiv & t_1 :: t_2 :: \cdots :: (\mathsf{sing}\ t_n) \end{array}$$

where :: is right-associative.

To clarify the terminology: a matrix with 3 rows and 4 columns is for us a two-*dimensional* array whose *sizes* are 3 and 4. Finite dimensional arrays are often represented by nested vectors and that is the approach we adopt as well. The challenge then is to ensure that the nested vectors do correspond to arrays, in other words that all entries in a vector have the same length. As we will see in the next chapter, we will use shape analysis to identify ill-formed arrays.

3.2 The Vec language

Vec is a simply typed language with a vector type constructor vec having the Size language in its core.

The type system of Vec is given by the following grammar

$$\begin{array}{lcl} \delta & ::= & \mathsf{nat} \mid \mathsf{bool} \mid \ldots \\ \tau & ::= & \delta \mid \mathsf{un} \mid \mathsf{sz} \mid \tau \times \tau \mid \mathsf{vec}\ \tau \\ \sigma & ::= & \tau \mid \sigma \times \sigma \mid \sigma \to \sigma\ . \end{array}$$

Remarks

1. The types δ are the *datum* types representing the basic data of the system, such as natural numbers, booleans and so on.

2. The types τ are the *data* types which include the datum types, the type sz of sizes, and are closed under finite products and vector constructions.

3. σ ranges over *phrase* types which include the data types and functions between them. The reason for distinguishing phrase types from data types is to disallow construction of types such as vectors of functions – as we will see in the next chapter, we need decidable equality on the

(shapes of) entries of vectors, and this would be problematic in the presence of functions.

A *ground* type is one which does not contain any arrow types. The *ground data-free* types, indicated by the meta-variable θ, are just products of copies of sz and un:

$$\theta ::= \mathsf{un} \mid \mathsf{sz} \mid \theta \times \theta \ .$$

As in SIZE, these are the types for which the equality combinator eq is introduced.

Finally, ς ranges over the *discrete* types generated by

$$\varsigma ::= \delta \mid \mathsf{un} \mid \varsigma \times \varsigma \mid \varsigma \to \varsigma \ .$$

Discrete types carry only trivial shape information.

un	:	un
$\mathsf{pair}^{\sigma,\sigma'}$	:	$\sigma \to \sigma' \to \sigma \times \sigma'$
$\mathsf{fst}^{\sigma,\sigma'}$	:	$\sigma \times \sigma' \to \sigma$
$\mathsf{snd}^{\sigma,\sigma'}$	:	$\sigma \times \sigma' \to \sigma'$
zero	:	sz
succ	:	$\mathsf{sz} \to \mathsf{sz}$
pred	:	$\mathsf{sz} \to \mathsf{sz}$
eq^{θ}	:	$\theta \to \theta \to \mathsf{sz}$
err^{σ}	:	σ
sing^{τ}	:	$\tau \to \mathsf{vec}\ \tau$
cons^{τ}	:	$\tau \to \mathsf{vec}\ \tau \to \mathsf{vec}\ \tau$
hd^{τ}	:	$\mathsf{vec}\ \tau \to \tau$
tl^{τ}	:	$\mathsf{vec}\ \tau \to \mathsf{vec}\ \tau$
length^{τ}	:	$\mathsf{vec}\ \tau \to \mathsf{sz}$

Figure 3.1: Typing of VEC combinators

The terms of the system are

$$t \ ::= \ x \mid c \mid \lambda x^{\sigma}.t \mid t\ t \mid \mathsf{rec}\ f^{\sigma}.\ t \mid \mathsf{ifs}\ t\ \mathsf{then}\ t\ \mathsf{else}\ t$$

where c is a combinator, the list of which is given in Fig 3.1. The combinators are either those of SIZE, and as such were discussed in the previous chapter, or the vector combinators from the previous section. Note that we have not included operations on datum types though in practice one would introduce them (and we will do so in Section 3.3). As in SIZE, we use sizes as conditions in shape conditionals, and the following terms as logical operations on sizes:

$$\begin{array}{lcl} \mathsf{and} & : & \mathsf{sz} \to \mathsf{sz} \to \mathsf{sz} \\ \mathsf{and} & \equiv & \lambda x.\lambda y.\mathsf{ifs}\ x\ \mathsf{then}\ y\ \mathsf{else}\ x \\ \mathsf{not} & : & \mathsf{sz} \to \mathsf{sz} \\ \mathsf{not} & \equiv & \lambda x.\mathsf{ifs}\ x\ \mathsf{then}\ \mathsf{succ}\ x\ \mathsf{else}\ x\ . \end{array}$$

Analogously as for SIZE we define the notions of value and result:

Definition 3.2.1 A *value* v is a term given by the following grammar

$$\begin{array}{rcl} v & ::= & x \mid c\ (\not\equiv \mathsf{err}) \mid \lambda x^{\sigma}.t \mid \mathsf{pair}\ v \mid\ \mathsf{pair}\ v\ v \mid \mathsf{succ}\ v \mid \mathsf{eq}\ v \\ & & \mid \mathsf{sing}\ v \mid \mathsf{cons}\ v \mid \mathsf{cons}\ v\ v\ . \end{array}$$

A result r is either a value or err.

As before, we will use v and r to denote values and results, respectively.

The type inference rules are given in Fig 3.2, and the reduction rules in Fig 3.3 (as in the case of SIZE, the reduction is lazy).

We will not prove the equivalents of most of the standard lemmas proved in Section 2.1 for the core lambda calculus as their proofs in VEC are analogous.

Theorem 3.2.2 $\to_{\mathsf{vc}}$ *is confluent.*

Proof. Given in Appendix A. The proof is analogous to that of the confluency of SIZE (Theorem 2.3.3). ∎

It can be easily seen that all SIZE terms are also terms of VEC. Moreover, as the following lemma shows, the reductions of SIZE terms are the same in both languages.

Lemma 3.2.3 Let t be a SIZE term. If $t \to_{\mathsf{vc}}^{*} t'$, then t' is a SIZE term. Moreover

$$t \to_{\mathsf{sz}}^{*} t' \quad \text{iff} \quad t \to_{\mathsf{vc}}^{*} t'\ .$$

Proof. By induction on the number of reduction steps in $t \to_{\mathsf{sz}}^{*} t'$. ∎

Variable	$x : \sigma \vdash x : \sigma$
Weakening	$\dfrac{\Gamma \vdash t : \sigma \quad x \notin Dom(\Gamma)}{\Gamma, x : \sigma' \vdash t : \sigma}$
Combinator	$\dfrac{c : \sigma}{\Gamma \vdash c : \sigma}$
Abs	$\dfrac{\Gamma, x : \sigma \vdash t : \sigma'}{\Gamma \vdash \lambda x^{\sigma}.t : \sigma \to \sigma'}$
App	$\dfrac{\Gamma \vdash t : \sigma' \to \sigma \quad \Gamma \vdash t' : \sigma'}{\Gamma \vdash t\ t' : \sigma}$
Rec	$\dfrac{\Gamma, f : \sigma \vdash t : \sigma}{\Gamma \vdash \mathsf{rec}\ f^{\sigma}.t : \sigma}$
Ifs	$\dfrac{\Gamma \vdash t : \mathsf{sz} \quad \Gamma \vdash t_1 : \sigma \quad \Gamma \vdash t_2 : \sigma}{\Gamma \vdash \mathsf{ifs}\ t\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2 : \sigma}$

Figure 3.2: Typing judgments for VEC

So SIZE really is a "sublanguage" of VEC, and reductions in both languages are the same (similarly for evaluation, see Lemma 3.2.7 below). Therefore, when there will be no danger of confusion, we may drop the subscripts in $\to_{\mathsf{sz}}$ and $\to_{\mathsf{vc}}$.

Lemma 3.2.4 Let v be a value.

- If $v : \mathsf{sz}$, then there exists n such that $v \equiv \tilde{\mathsf{n}}$.
- If $v : \sigma_1 \times \sigma_2$, then $v \equiv \langle v_1, v_2 \rangle$ for some values $v_1 : \sigma_1$ and $v_2 : \sigma_2$.
- If $v : \mathsf{vec}\ \tau$, then there exist values $v_1, \ldots, v_n$ of type τ such that $v \equiv [v_1, \ldots, v_n]$.

Proof. Follows immediately from the definition. ∎

$(\lambda x.t)\ v$	$\rightarrow_{\mathsf{vc}}$	$t[v/x]$
$\mathsf{fst}\ \langle t_1, v\rangle$	$\rightarrow_{\mathsf{vc}}$	t_1
$\mathsf{snd}\ \langle v, t_2\rangle$	$\rightarrow_{\mathsf{vc}}$	t_2
$\mathsf{pred\ zero}$	$\rightarrow_{\mathsf{vc}}$	err
$\mathsf{pred}\ (\mathsf{succ}\ t)$	$\rightarrow_{\mathsf{vc}}$	t
$\mathsf{eq\ zero}\ t$	$\rightarrow_{\mathsf{vc}}$	t
$\mathsf{eq}\ (\mathsf{succ}\ t)\ \mathsf{zero}$	$\rightarrow_{\mathsf{vc}}$	$\mathsf{succ}\ t$
$\mathsf{eq}\ (\mathsf{succ}\ t)\ (\mathsf{succ}\ t')$	$\rightarrow_{\mathsf{vc}}$	$\mathsf{eq}\ t\ t'$
$\mathsf{eq\ un\ un}$	$\rightarrow_{\mathsf{vc}}$	zero
$\mathsf{eq}\ \langle t_1, t_2\rangle\ \langle s_1, s_2\rangle$	$\rightarrow_{\mathsf{vc}}$	$\mathsf{and}\ (\mathsf{eq}\ t_1\ t_2)\ (\mathsf{eq}\ s_1\ s_2)$
$\mathsf{hd}\ [t]$	$\rightarrow_{\mathsf{vc}}$	t
$\mathsf{hd}\ (t_1 :: v)$	$\rightarrow_{\mathsf{vc}}$	t_1
$\mathsf{tl}\ [t]$	$\rightarrow_{\mathsf{vc}}$	err
$\mathsf{tl}\ (v :: t)$	$\rightarrow_{\mathsf{vc}}$	t
$\mathsf{length}\ [t]$	$\rightarrow_{\mathsf{vc}}$	$\mathsf{succ\ zero}$
$\mathsf{length}\ (v :: t)$	$\rightarrow_{\mathsf{vc}}$	$\mathsf{succ}\ (\mathsf{length}\ t)$
$\mathsf{rec}\ f.t$	$\rightarrow_{\mathsf{vc}}$	$t[\mathsf{rec}\ f.t/f]$
$\mathsf{ifs\ zero\ then}\ t_1\ \mathsf{else}\ t_2$	$\rightarrow_{\mathsf{vc}}$	t_1
$\mathsf{ifs\ succ}\ v\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2$	$\rightarrow_{\mathsf{vc}}$	t_2

Figure 3.3: Base reductions for VEC

$$\frac{r \text{ is a result}}{r \Rightarrow_{\mathsf{vc}} r} \qquad \frac{t_1 \Rightarrow_{\mathsf{vc}} \lambda x.t \quad t_2 \Rightarrow_{\mathsf{vc}} v \quad t[v/x] \Rightarrow_{\mathsf{vc}} r}{t_1\ t_2 \Rightarrow_{\mathsf{vc}} r}$$

$$\frac{t_1 \Rightarrow_{\mathsf{vc}} \mathsf{err}}{t_1\ t_2 \Rightarrow_{\mathsf{vc}} \mathsf{err}} \qquad \frac{t_1 \Rightarrow_{\mathsf{vc}} v \quad t_2 \Rightarrow_{\mathsf{vc}} \mathsf{err}}{t_1\ t_2 \Rightarrow_{\mathsf{vc}} \mathsf{err}}$$

$$\frac{t \Rightarrow_{\mathsf{vc}} \mathsf{zero} \quad t_1 \Rightarrow_{\mathsf{vc}} r}{\mathsf{ifs}\ t\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2 \Rightarrow_{\mathsf{vc}} r} \qquad \frac{t \Rightarrow_{\mathsf{vc}} \mathsf{succ}\ v \quad t_2 \Rightarrow_{\mathsf{vc}} r}{\mathsf{ifs}\ t\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2 \Rightarrow_{\mathsf{vc}} r}$$

$$\frac{t \Rightarrow_{\mathsf{vc}} \mathsf{err}}{\mathsf{ifs}\ t\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2 \Rightarrow_{\mathsf{vc}} \mathsf{err}} \qquad \frac{t[\mathsf{rec}\ f.t/f] \Rightarrow_{\mathsf{vc}} r}{\mathsf{rec}\ f.t \Rightarrow_{\mathsf{vc}} r}$$

Figure 3.4: Operational semantics of VEC– Part 1

$$\frac{t_1 \Rightarrow_{\mathsf{vc}} \mathsf{pair} \quad t_2 \Rightarrow_{\mathsf{vc}} v}{t_1\ t_2 \Rightarrow_{\mathsf{vc}} \mathsf{pair}\ v} \qquad \frac{t_1 \Rightarrow_{\mathsf{vc}} \mathsf{pair}\ v_1 \quad t_2 \Rightarrow_{\mathsf{vc}} v_2}{t_1\ t_2 \Rightarrow_{\mathsf{vc}} \langle v_1, v_2 \rangle}$$

$$\frac{t_1 \Rightarrow_{\mathsf{vc}} \mathsf{fst} \quad t_2 \Rightarrow_{\mathsf{vc}} \langle v_1, v_2 \rangle}{t_1\ t_2 \Rightarrow_{\mathsf{vc}} v_1} \qquad \frac{t_1 \Rightarrow_{\mathsf{vc}} \mathsf{snd} \quad t_2 \Rightarrow_{\mathsf{vc}} \langle v_1, v_2 \rangle}{t_1\ t_2 \Rightarrow_{\mathsf{vc}} v_2}$$

$$\frac{t_1 \Rightarrow_{\mathsf{vc}} \mathsf{succ} \quad t_2 \Rightarrow_{\mathsf{vc}} v}{t_1\ t_2 \Rightarrow_{\mathsf{vc}} \mathsf{succ}\ v} \qquad \frac{t_1 \Rightarrow_{\mathsf{vc}} \mathsf{pred} \quad t_2 \Rightarrow_{\mathsf{vc}} \mathsf{succ}\ v}{t_1\ t_2 \Rightarrow_{\mathsf{vc}} v}$$

$$\frac{t_1 \Rightarrow_{\mathsf{sz}} \mathsf{pred} \quad t_2 \Rightarrow_{\mathsf{sz}} \mathsf{zero}}{t_1\ t_2 \Rightarrow_{\mathsf{sz}} \mathsf{err}} \qquad \frac{t_1 \Rightarrow_{\mathsf{sz}} \mathsf{eq} \quad t_2 \Rightarrow_{\mathsf{sz}}\ v}{t_1\ t_2 \Rightarrow_{\mathsf{sz}} \mathsf{eq}\ v}$$

$$\frac{t_1 \Rightarrow_{\mathsf{vc}} \mathsf{eq}\ \mathsf{zero} \quad t_2 \Rightarrow_{\mathsf{vc}} v}{t_1\ t_2 \Rightarrow_{\mathsf{vc}} v} \qquad \frac{t_1 \Rightarrow_{\mathsf{vc}} \mathsf{eq}\ (\mathsf{succ}\ v) \quad t_2 \Rightarrow_{\mathsf{vc}} \mathsf{zero}}{t_1\ t_2 \Rightarrow_{\mathsf{vc}} (\mathsf{succ}\ v}$$

$$\frac{t_1 \Rightarrow_{\mathsf{vc}} \mathsf{eq}\ (\mathsf{succ}\ v) \qquad t_2 \Rightarrow_{\mathsf{vc}} \mathsf{succ}\ v' \qquad \mathsf{eq}\ v\ v' \Rightarrow_{\mathsf{vc}} v''}{t_1\ t_2 \Rightarrow_{\mathsf{vc}} v''} \qquad \frac{t_1 \Rightarrow_{\mathsf{vc}} \mathsf{eq}\ \mathsf{un} \quad t_2 \Rightarrow_{\mathsf{vc}} \mathsf{un}}{t_1\ t_2 \Rightarrow_{\mathsf{vc}} \mathsf{zero}}$$

$$\frac{t_1 \Rightarrow_{\mathsf{vc}} \mathsf{eq}\ \langle v_1, v_2 \rangle \qquad t_2 \Rightarrow_{\mathsf{vc}} \langle v_1', v_2' \rangle \qquad \mathsf{and}\ (\mathsf{eq}\ v_1\ v_1')\ (\mathsf{eq}\ v_2\ v_2') \Rightarrow_{\mathsf{vc}} v''}{t_1\ t_2 \Rightarrow_{\mathsf{vc}} v''} \qquad \frac{t_1 \Rightarrow_{\mathsf{vc}} \mathsf{sing} \quad t_2 \Rightarrow_{\mathsf{vc}} v}{t_1\ t_2 \Rightarrow_{\mathsf{vc}} \mathsf{sing}\ v}$$

$$\frac{t_1 \Rightarrow_{\mathsf{vc}} \mathsf{cons} \quad t_2 \Rightarrow_{\mathsf{vc}} v}{t_1\ t_2 \Rightarrow_{\mathsf{vc}} \mathsf{cons}\ v} \qquad \frac{t_1 \Rightarrow_{\mathsf{vc}} \mathsf{cons}\ v_1 \quad t_2 \Rightarrow_{\mathsf{vc}} v}{t_1\ t_2 \Rightarrow_{\mathsf{vc}} \mathsf{cons}\ v_1\ v}$$

$$\frac{t_1 \Rightarrow_{\mathsf{vc}} \mathsf{hd} \quad t_2 \Rightarrow_{\mathsf{vc}} [v]}{t_1\ t_2 \Rightarrow_{\mathsf{vc}} v} \qquad \frac{t_1 \Rightarrow_{\mathsf{vc}} \mathsf{hd} \quad t_2 \Rightarrow_{\mathsf{vc}} v_1 :: v}{t_1\ t_2 \Rightarrow_{\mathsf{vc}} v_1}$$

$$\frac{t_1 \Rightarrow_{\mathsf{vc}} \mathsf{tl} \quad t_2 \Rightarrow_{\mathsf{vc}} [v]}{t_1\ t_2 \Rightarrow_{\mathsf{vc}} \mathsf{err}} \qquad \frac{t_1 \Rightarrow_{\mathsf{vc}} \mathsf{tl} \quad t_2 \Rightarrow_{\mathsf{vc}} v_1 :: v}{t_1\ t_2 \Rightarrow_{\mathsf{vc}} v}$$

$$\frac{t_1 \Rightarrow_{\mathsf{vc}} \mathsf{length} \quad t_2 \Rightarrow_{\mathsf{vc}} [v]}{t_1\ t_2 \Rightarrow_{\mathsf{vc}} \mathsf{succ}\ \mathsf{zero}} \qquad \frac{t_1 \Rightarrow_{\mathsf{vc}} \mathsf{length} \qquad t_2 \Rightarrow_{\mathsf{vc}} v_1 :: v \qquad \mathsf{length}\ v \Rightarrow_{\mathsf{vc}} v'}{t_1\ t_2 \Rightarrow_{\mathsf{vc}} \mathsf{succ}\ v'}$$

Figure 3.5: Operational semantics of Vec– Part 2

The operational semantics of VEC, given in Fig 3.4 and Fig 3.5, is an extension of that of SIZE.

Lemma 3.2.5 *Let t be a closed term.*

1. *If $t \Rightarrow_{\mathsf{vc}} r$, then r is a result.*
2. *If $t \Rightarrow_{\mathsf{vc}} r_1$ and $t \Rightarrow_{\mathsf{vc}} r_2$, then $r_1 \equiv r_2$.*
3. *If $t \Rightarrow_{\mathsf{vc}} v$ (where $v \not\equiv \mathsf{err}$), then $t \rightarrow^*_{\mathsf{vc}} v$.*
4. *If $\vdash t : \sigma$ and $t \Rightarrow_{\mathsf{vc}} r$, then $\vdash r : \sigma$.*

Proof. Analogous to the proofs of the same properties of the operational semantics of SIZE. ∎

Theorem 3.2.6 (Soundness theorem). *Let s be a closed term. If $s =_{\mathsf{vc}} s'$ then $s \approx_{\mathsf{vc}} s'$.*

Proof. Analogous to the proof of the soundness theorem for SIZE. ∎

SIZE terms evaluate in the same way in SIZE and VEC, as the following lemma shows.

Lemma 3.2.7 Let t be a closed SIZE term. Whenever $t \Rightarrow_{\mathsf{vc}} r$, then r is a SIZE term and moreover $t \Rightarrow_{\mathsf{sz}} r$ iff $t \Rightarrow_{\mathsf{vc}} r$.

Proof. We can immediately see that all the evaluation rules of $\Rightarrow_{\mathsf{sz}}$ are among those of $\Rightarrow_{\mathsf{vc}}$. ∎

Since evaluation in VEC is the same as in SIZE, we may drop the subscripts in $\Rightarrow_{\mathsf{sz}}$ and $\Rightarrow_{\mathsf{vc}}$.

3.3 Expressive power

We will now show that the Vec language, as defined in the previous section, is powerful enough to represent many commonly used array operations, in particular many of the usual array indexing operations, second-order array operations and those of linear algebra.

The following syntactic sugar will be useful:

$$\mathsf{is_sing}\ x \ \equiv\ \mathsf{pred}\ (\mathsf{length}\ x)\ .$$

It is easy to see that is_sing t evaluates to zero if and only if t evaluates to a singleton.

3.3.1 Array indexing

Arrays in Vec are just nested vectors, and as such are built up using the sing and cons combinators. This may seem a strange way to construct arrays, as in most (imperative) languages arrays are created in "one step" and their entries can then be accessed and modified, using operations such as the following:

$$\begin{array}{lll} \mathsf{make}\ n\ a & \Rightarrow\ [\underbrace{a, a, \ldots, a}_{n\text{-times}}] & \text{if } n \geq 1 \\ \mathsf{read}\ m\ [a_1, \ldots, a_n] & \Rightarrow\ a_m \quad \text{if } m \leq n & \\ \mathsf{write}\ [a_1, \ldots, a_n]\ m\ b & \Rightarrow\ [a_1, \ldots, a_{m-1}, b, a_{m+1}, \ldots, a_n] & \text{if } m \leq n \end{array}$$

(with the operations resulting in an error when the side conditions are not satisfied). We begin our account of expressivity of Vec by defining the operations above.

Since the indices (the m and n arguments in the examples above) refer to the lengths (i.e. sizes) of vectors, they have to be represented in Vec by terms of type sz, rather than nat. The types of the operations are thus

$$\begin{array}{lll} \mathsf{make} & : & \mathsf{sz} \to \tau \to \mathsf{vec}\ \tau \\ \mathsf{read} & : & \mathsf{sz} \to \mathsf{vec}\ \tau \to \tau \\ \mathsf{write} & : & \mathsf{vec}\ \tau \to \mathsf{sz} \to \tau \to \mathsf{vec}\ \tau\ . \end{array}$$

The terms representing these operations are given below.

$$\begin{aligned}
\mathsf{make} &\equiv \mathsf{rec}\ f.\lambda n, x.\mathsf{ifs\ pred}\ n\ \mathsf{then}\ [x] \\
&\qquad\qquad\qquad\qquad \mathsf{else}\ x :: (f\ (\mathsf{pred}\ n)\ x) \\
\mathsf{read} &\equiv \mathsf{rec}\ f.\lambda n, x.\mathsf{ifs\ pred}\ n\ \mathsf{then\ hd}\ x \\
&\qquad\qquad\qquad\qquad \mathsf{else}\ f\ (\mathsf{pred}\ n)\ (\mathsf{tl}\ x) \\
\mathsf{write} &\equiv \mathsf{rec}\ f.\lambda x, n, y.\mathsf{ifs\ pred}\ n \\
&\qquad\quad \mathsf{then\ ifs\ is_sing}\ x\ \mathsf{then}\ [y] \\
&\qquad\qquad\qquad\qquad \mathsf{else}\ y :: (\mathsf{tl}\ x) \\
&\qquad\quad \mathsf{else}\ (\mathsf{hd}\ x) :: (f\ (\mathsf{tl}\ x)\ (\mathsf{pred}\ n)\ y)\ .
\end{aligned}$$

We define $\mathsf{mat\ nat} \equiv \mathsf{vec\ (vec\ nat)}$ to be the type of matrices of natural numbers, represented as columns of rows. Then we can build matrices using the operations given above

$$\begin{aligned}
\mathsf{mat1} &\equiv \mathsf{make}\ \tilde{}2\ (\mathsf{make}\ \tilde{}3\ 1) \\
\mathsf{mat2} &\equiv \mathsf{make}\ \tilde{}3\ (\mathsf{make}\ \tilde{}1\ 2)
\end{aligned}$$

where $1, 2 : \mathsf{nat}$. Then

$$\begin{aligned}
\mathsf{mat1} &\Rightarrow [[1,1,1],[1,1,1]] \\
\mathsf{mat2} &\Rightarrow [[2],[2],[2]]\ .
\end{aligned}$$

It is easy to see that, when applied, the operations evaluate the way we expect, including evaluating to err when the array references are out of bounds:

$$\begin{aligned}
\mathsf{read}\ \tilde{}3\ (\mathsf{read}\ \tilde{}2\ \mathsf{mat1}) &\Rightarrow 1 \\
\mathsf{read}\ \tilde{}3\ (\mathsf{read}\ \tilde{}2\ \mathsf{mat2}) &\Rightarrow \mathsf{err}
\end{aligned}$$

(similarly for write).

Note that our decision to use the sz type to represent array indices was not merely our whim – it is easy to see that the types of Vec constructs do not allow us to express, say, the operation make with its type being $\mathsf{nat} \rightarrow \tau \rightarrow \mathsf{vec}\ \tau$. The main reason is that the condition of the ifs conditional has to be of type sz, and there is no way in Vec to convert a term of type nat to its corresponding size. The absence of such natural numbers-to-sizes conversion operation is not an oversight on our part, such an operation, expressed in any way whatsoever, would be an unsurpassable barrier to successful shape analysis (as discussed in detail in the Chapter 6), and would therefore thwart the main objective of Vec.

3.3.2 Second order vector operations

Let us now turn our attention to some second order vector operations. We claimed, in the first section of this chapter, that there was no need to introduce additional vector combinators, such as map or fold, as VEC would turn out to be powerful enough to express them. We will now present the VEC representations of these second order operations (and of append and zip). These operations typically evaluate according to the following rules:

$$\begin{aligned}
\mathsf{map}\ f\ [a_1,\ldots,a_n] &\Rightarrow [f\ a_1,\ldots,f\ a_n]\\
\mathsf{fold}\ f\ x\ [a_1,\ldots,a_n] &\Rightarrow f\ \langle a_1,\ldots f\ \langle a_n,x\rangle\ldots\rangle\\
\mathsf{append}\ [a_1,\ldots,a_m]\ [b_1,\ldots,b_n] &\Rightarrow [a_1,\ldots,a_m,b_1,\ldots,b_n]\\
\mathsf{zip}\ [a_1,\ldots,a_n]\ [b_1,\ldots,b_n] &\Rightarrow [\langle a_1,b_1\rangle,\ldots,\langle a_n,b_n\rangle]
\end{aligned}$$

(where zip applied to two vectors of unequal lengths results in an error). The types of their VEC representations therefore are

$$\begin{aligned}
&\mathsf{map} &&: (\tau\to\tau')\to\mathsf{vec}\ \tau\to\mathsf{vec}\ \tau'\\
&\mathsf{fold} &&: (\tau\times\sigma\to\sigma)\to\sigma\to\mathsf{vec}\ \tau\to\sigma\\
&\mathsf{append} &&: \mathsf{vec}\ \tau\to\mathsf{vec}\ \tau\to\mathsf{vec}\ \tau\\
&\mathsf{zip} &&: \mathsf{vec}\ \tau\to\mathsf{vec}\ \tau'\to\mathsf{vec}\ (\tau\times\tau')\ .
\end{aligned}$$

and the terms representing them are given below.

$$\begin{aligned}
\mathsf{map} \equiv\ & \mathsf{rec}\ g.\lambda f,x.\mathsf{ifs}\ \mathsf{is_sing}\ x\\
&\quad \mathsf{then}\ [f\ (\mathsf{hd}\ x)]\\
&\quad \mathsf{else}\ (f\ (\mathsf{hd}\ x)) :: (g\ f\ (\mathsf{tl}\ x))\\
\mathsf{fold} \equiv\ & \mathsf{rec}\ g.\lambda f,x,y.\mathsf{ifs}\ \mathsf{is_sing}\ y\\
&\quad \mathsf{then}\ f\ \langle \mathsf{hd}\ y,x\rangle\\
&\quad \mathsf{else}\ f\ \langle \mathsf{hd}\ y, g\ f\ x\ (\mathsf{tl}\ y)\rangle\\
\mathsf{append} \equiv\ & \mathsf{rec}\ g.\lambda x,y.\mathsf{ifs}\ \mathsf{is_sing}\ x\\
&\quad \mathsf{then}\ (\mathsf{hd}\ x) :: y\\
&\quad \mathsf{else}\ (\mathsf{hd}\ x) :: (g\ (\mathsf{tl}\ x)\ y)\\
\mathsf{zip} \equiv\ & \mathsf{rec}\ g.\lambda x,y.\mathsf{ifs}\ \mathsf{is_sing}\ x\\
&\quad \mathsf{then}\ \mathsf{ifs}\ \mathsf{is_sing}\ y\ \mathsf{then}\ [\langle \mathsf{hd}\ x,\mathsf{hd}\ y\rangle]\ \mathsf{else}\ \mathsf{err}\\
&\quad \mathsf{else}\ \langle \mathsf{hd}\ x,\mathsf{hd}\ y\rangle :: (g\ (\mathsf{tl}\ x)\ (\mathsf{tl}\ y))
\end{aligned}$$

Their evaluations give the intended results, as in

$$\begin{aligned} &\mathsf{zip}\ (\mathsf{read}\ \tilde{}2\ \mathsf{mat1})\ \mathsf{mat2} \Rightarrow [\langle 1, [2]\rangle, \langle 1, [2]\rangle, \langle 1, [2]\rangle] \\ &\mathsf{zip}\ \mathsf{mat1}\ \mathsf{mat2} \Rightarrow \mathsf{err}\ . \end{aligned}$$

We will in later examples use the uncurried versions of the above terms, and we shall use primes to denote them, as in

$$\begin{aligned} \mathsf{zip}' &: (\mathsf{vec}\ \tau) \times (\mathsf{vec}\ \tau') \to \mathsf{vec}\ (\tau \times \tau') \\ \mathsf{zip}' &\equiv \lambda x.\mathsf{zip}\ (\mathsf{fst}\ x)\ (\mathsf{snd}\ x)\ . \end{aligned}$$

3.3.3 Linear algebra

We conclude the chapter with several operations over matrices of natural numbers. We begin with matrix transposition:

$$\begin{aligned} \mathsf{transpose} &: \mathsf{mat}\ \tau \to \mathsf{mat}\ \tau \\ \mathsf{transpose} &\equiv \mathsf{rec}\ f.\lambda x.\mathsf{ifs}\ \mathsf{is_sing}\ (\mathsf{hd}\ x) \\ &\qquad\quad \mathsf{then}\ [\mathsf{map}\ \mathsf{hd}\ x] \\ &\qquad\quad \mathsf{else}\ (\mathsf{map}\ \mathsf{hd}\ x) :: (f\ (\mathsf{map}\ \mathsf{tl}\ x)) \end{aligned}$$

We would now like to define matrix multiplication, as an example of a linear algebra operation only defined on matrices of matching sizes. To do this we have to introduce some elementary algebraic operations on natural numbers not expressible in the core VEC. So in the following we assume we have combinators

$$+, -, *, / : \mathsf{nat} \times \mathsf{nat} \to \mathsf{nat}$$

with the usual evaluation rules (with / representing the integer division).

Recall that the scalar product of two vectors of natural numbers is given by

$$\mathsf{scalar}\ [a_1, \ldots, a_n]\ [b_1, \ldots, b_n] \Rightarrow a_1 * b_1 + \ldots + a_n * b_n\ .$$

scalar is the curried version of the term scalar′ given by

$$\begin{aligned} \mathsf{scalar}' &: (\mathsf{vec}\ \mathsf{nat}) \times (\mathsf{vec}\ \mathsf{nat}) \to \mathsf{nat} \\ \mathsf{scalar}' &\equiv (\mathsf{fold}\ +\ 0) \circ (\mathsf{map}\ *) \circ \mathsf{zip}'\ . \end{aligned}$$

where $\circ$ represents the usual operation of function composition, that is $t_1 \circ t_2$ is syntactic sugar for $\lambda x.t_1\ (t_2\ x)$.

Finally, matrix multiplication is given by

$$\begin{aligned}\mathsf{multiply} &: \mathsf{mat\ nat} \to \mathsf{mat\ nat} \to \mathsf{mat\ nat} \\ \mathsf{multiply} &\equiv \lambda x, y.\ \mathsf{map}\ (\lambda z. \\ &\quad \mathsf{map}\ (\mathsf{scalar}\ z)\ (\mathsf{transpose}\ y))\ x\ .\end{aligned}$$

When we multiply two matrices we get the expected results, as in

$$\begin{aligned}\mathsf{multiply\ mat1\ mat2} &\Rightarrow [[6],[6]] \\ \mathsf{multiply\ mat2\ mat1} &\Rightarrow \mathsf{err}\ .\end{aligned}$$

Chapter 4

Shape analysis

This chapter introduces shapes and shape analysis and proves their main properties, and as such represents the core part of the thesis. We begin the chapter by defining a translation # from VEC to SIZE which maps a term in VEC to its shape in SIZE. Shape of a term will typically be much simpler than the term itself as the shape translation ignores all data – for example, the shape of a matrix will be just a triple of consisting of the pair of its sizes and the shape of its entries. Consequently, the evaluation of a shape will often be much faster than that of the original term. This whole process, mapping a term to its shape and then evaluating this shape, will be referred to as shape analysis. As we have already stressed, we will concentrate on using shape analysis for error detection and, in the final section of this chapter, we will show that shape analysis reveals all shape errors in a term. Another important result will show that shape analysis of a term terminates whenever its evaluation does, thus demonstrating the viability of shape analysis as a new kind of static program analysis.

The chapter is structured as follows. The first section introduces the shape translation # from VEC to SIZE. In the second section we then apply shape analysis to a variety of array-based programs. Finally, in the third, rather technical section we prove some of the important properties of shape analysis.

As we have showed in the previous chapter, SIZE is a sublanguage of VEC, and we will not therefore distinguish operations on SIZE terms in SIZE from those in VEC. Unless stated otherwise, we will assume that all operations

are taking place in VEC.

4.1 The shape translation

We begin by defining the shape translation # from VEC to SIZE. This translation acts on both the types and the terms of VEC.

Definition 4.1.1 The *shape translation # of* VEC *types* is a SIZE type defined inductively by

$$
\begin{array}{lcl}
\#\delta & \equiv & \mathsf{un} \\
\#\mathsf{un} & \equiv & \mathsf{un} \\
\#\mathsf{sz} & \equiv & \mathsf{sz} \\
\#(\mathsf{vec}\ \tau) & \equiv & \mathsf{sz} \times \#\tau \\
\#(\sigma \times \sigma') & \equiv & \#\sigma \times \#\sigma' \\
\#(\sigma \to \sigma') & \equiv & \#\sigma \to \#\sigma' \ .
\end{array}
$$

We say that $\#\sigma$ is the *shape* of σ.

Remarks

1. As there will be no danger of confusion, we shall use the same symbol # to denote the shape translation of both the types and terms (see below). # will always bind tightest.

2. Note that # is the identity on the types of SIZE, indicating that SIZE terms are "pure shapes".

3. The shape translation preserves the structure of a type, with two important exceptions: it ignores datum types, i.e. their shapes are the unit, and it maps the type $\mathsf{vec}\ \tau$ to $\mathsf{sz} \times \#\tau$. Thus a shape of a vector is a pair of its length (a size) and the uniform shape of its entries.

Definition 4.1.2 The *shape translation # of* VEC *terms* is a SIZE term defined inductively using the rules in Fig 4.1. We say that $\#t$ is the *shape* of t.

$$
\begin{array}{lll}
\#x^\sigma & \equiv & x^{\#\sigma} \\
\#\mathsf{sing} & \equiv & \lambda x.\langle \mathsf{succ\ zero}, x\rangle \\
\#\mathsf{cons} & \equiv & \lambda x, y.\mathsf{ifs\ eq}\ x\ (\mathsf{snd}\ y) \\
 & & \qquad \mathsf{then}\ \langle \mathsf{succ}\ (\mathsf{fst}\ y), \mathsf{snd}\ y\rangle\ \mathsf{else\ err} \\
\#\mathsf{hd} & \equiv & \mathsf{snd} \\
\#\mathsf{tl} & \equiv & \lambda x.\mathsf{ifs\ pred}\ (\mathsf{fst}\ x)\ \mathsf{then\ err} \\
 & & \qquad \mathsf{else}\ \langle \mathsf{pred}\ (\mathsf{fst}\ x), \mathsf{snd}\ x\rangle \\
\#\mathsf{length} & \equiv & \mathsf{fst} \\
\#c & \equiv & c \quad \text{for other combinators } c \\
\#(\lambda x^\sigma.t) & \equiv & \lambda x^{\#\sigma}.\#t \\
\#(t_1\ t_2) & \equiv & \#t_1\ \#t_2 \\
\#(\mathsf{rec}\ f.t) & \equiv & \mathsf{rec}\ f.\#t \\
\#(\mathsf{ifs}\ t\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2) & \equiv & \mathsf{ifs}\ \#t\ \mathsf{then}\ \#t_1\ \mathsf{else}\ \#t_2
\end{array}
$$

Figure 4.1: Shape translation of terms

Remarks

1. The process of taking the shape of a term and evaluating it will be loosely referred to as *shape analysis*.

2. As before, we can see that # is the identity on SIZE.

3. The shapes of operations on datum types, such as those introduced in Section 3.3, will be given in Section 4.2.

4. The most interesting part of Fig 4.1 is the shapes of vector combinators. As we have already mentioned, the shape of a vector is a pair – its length and the shape of its entries. Thus, for example, the shapes of length and hd are the first and second projection, respectively. The shape of cons is more complicated, as it has to ensure that all entries in a vector have the same shape. Thus shape analysis of cons t_1 t checks whether $\#t_1$ and snd $\#t$ (the shape of the entries of t) are equal, and returns error otherwise. In this way shape analysis rejects any non-regular vectors, that is nested vectors which are not arrays.

5. Being able to check for the equality of the shapes of entries of vectors has been the rationale behind preventing the formation of, for example, the type of vectors of functions (achieved by stratifying VEC types into data and phrase types).

6. We can now also understand the decision to introduce the singleton, rather than the customary empty vector, as the elementary vector combinator. Since an empty vector does not have any entries, shape analysis of a term such as cons t nil would be problematic (though not impossible, as discussed in Section 6.4). So as not to complicate matters at this point, we have opted for preventing the formation of empty vectors.

We extend # to operate on typing contexts in the obvious way:

$$\#\{x_1 : \sigma_1, \ldots, x_n : \sigma_n\} = \{x_1 : \#\sigma_1, \ldots, x_n : \#\sigma_n\} \ .$$

The following lemma shows that # respects typing.

Lemma 4.1.3 If $\Gamma \vdash t : \sigma$ then $\#\Gamma \vdash \#t : \#\sigma$.

Proof. By induction on the height of derivation of $\Gamma \vdash t : \sigma$. ∎

As we have said before, we want to concentrate on error detection. The terminology introduced in the following definition will thus be often used.

Definition 4.1.4 We say that *the evaluation of t results in a shape error* whenever $t \Rightarrow$ err. If $\#t \Rightarrow$ err, we may say that shape analysis of t *reveals* a shape error, or that t is *ill shaped*. It is *well shaped* otherwise.

Remark It is important to realise that not every error can be viewed as shape error. For example, in most languages (integer) division by zero is treated as error, but this kind of error, regardless of the way it is handled in the operational semantics, cannot be detected by shape analysis, since shape analysis does not distinguish zero from other integers (as they all have the same shape, the unit).

Before we prove other claims about shape analysis and its ability to reveal shape errors in Section 4.3, let us first illustrate its actions in the following section.

4.2 Examples

We will now use the techniques of the previous section to analyse the VEC terms defined in Section 3.3. Before we begin, we have to extend # to cover the operations on datum types introduced in that section, namely the algebraic operations on natural numbers. As it turns out there is essentially only one way of doing that without compromising the important properties of shape analysis (as described in the next section), and that is by letting

$$\begin{aligned} \#n &\equiv \mathsf{un} \\ \#c &\equiv \lambda x.\mathsf{un} \end{aligned}$$

where $n \in \{0, 1, \ldots\} : \mathsf{nat}$ and $c \in \{+, -, *, /\} : \mathsf{nat} \times \mathsf{nat} \to \mathsf{nat}$. Note that we then have

$$\#(/\ \langle 1, 0\rangle) \Rightarrow \mathsf{un}$$

regardless of the way division by 0 is treated in the operational semantics.

4.2.1 Vectors

We start with examples of shape analysis of vectors of natural numbers:

$$\begin{aligned} &\#[1,1,1] &&\Rightarrow \langle \tilde{}3, \mathsf{un}\rangle \\ &\#[1,2,3] &&\Rightarrow \langle \tilde{}3, \mathsf{un}\rangle \\ &\#[\langle 1,1\rangle, \langle 2,2\rangle, \langle 3,3\rangle] &&\Rightarrow \langle \tilde{}3, \langle \mathsf{un}, \mathsf{un}\rangle\rangle \\ &\#[[1,1],[2,2],[3,3]] &&\Rightarrow \langle \tilde{}3, \langle \tilde{}2, \mathsf{un}\rangle\rangle \\ &\#[[1],[2,2],[3,3]] &&\Rightarrow \mathsf{err}\ . \end{aligned}$$

Shape analysis of the term $\mathsf{hd}\ [[1],[2,2],[3,3]]$ illustrates the fact that even terms whose evaluation does not result in a shape error may be ill shaped:

$$\begin{aligned} &\mathsf{hd}\ [[1],[2,2],[3,3]] &&\Rightarrow [1] \\ &\#(\mathsf{hd}\ [[1],[2,2],[3,3]]) &&\Rightarrow \mathsf{err}\ . \end{aligned}$$

On vectors of sizes, shape analysis gives the following results:

$$\begin{aligned} &\#[\tilde{}1, \tilde{}1, \tilde{}1] &&\Rightarrow \langle \tilde{}3, \tilde{}1\rangle \\ &\#[\langle \tilde{}1, 1\rangle, \langle \tilde{}1, 2\rangle, \langle \tilde{}1, 3\rangle] &&\Rightarrow \langle \tilde{}3, \langle \tilde{}1, \mathsf{un}\rangle\rangle \\ &\#[\tilde{}1, \tilde{}2, \tilde{}3] &&\Rightarrow \mathsf{err}\ . \end{aligned}$$

Note in particular the last example. Since the shape of a size is the size itself, a vector with different sizes as entries is ill shaped.

4.2.2 Array indexing

Let us now look at shape analysis of the array operations defined in Section 3.3. As the shapes of the operations make, read and write themselves are quite complex (and thus not very enlightening), we present here just the shape of make:

$$\begin{aligned}\#\mathsf{make} \equiv\ & \mathsf{rec}\ f.\ \lambda n, x.\mathsf{ifs\ pred}\ n\ \mathsf{then}\ (\lambda x.\langle \mathsf{succ\ zero}, x\rangle)\ x \\ & \mathsf{else}\ (\lambda x, y.\mathsf{ifs\ eq}\ x\ (\mathsf{snd}\ y)\ \mathsf{then}\ \langle \mathsf{succ}\ (\mathsf{fst}\ y), \mathsf{snd}\ y\rangle\ \mathsf{else\ err}) \\ & \quad x\ (f\ (\mathsf{pred}\ n)\ x)\ .\end{aligned}$$

A careful study of the above shape reveals that it is operationally equivalent to

$$\lambda n, x.\langle n, x\rangle\ .$$

The shapes of matrices then come out the way we expect:

$$\begin{aligned}\#\mathsf{mat1} &\Rightarrow \langle \tilde{}2, \langle \tilde{}3, \mathsf{un}\rangle\rangle \\ \#\mathsf{mat2} &\Rightarrow \langle \tilde{}3, \langle \tilde{}1, \mathsf{un}\rangle\rangle\ .\end{aligned}$$

Thus we can see that the number of rows of a matrix t is fst $\#t$ and the number of its columns is fst (snd $\#t$).

The array access errors arising from incorrect applications of read or write are revealed by shape analysis, as in

$$\begin{aligned}\#(\mathsf{read}\ \tilde{}3\ (\mathsf{read}\ \tilde{}2\ \mathsf{mat1})) &\Rightarrow \mathsf{un} \\ \#(\mathsf{read}\ \tilde{}3\ (\mathsf{read}\ \tilde{}2\ \mathsf{mat2})) &\Rightarrow \mathsf{err}\ .\end{aligned}$$

4.2.3 Second order vector operations

The comment about the complexities of the shapes of array-indexing operations is true also for operations such as map and fold. It is therefore more revealing to analyse applied operations, as in

$$\begin{aligned}\#(\mathsf{map}\ (\lambda x.x)\ [1,2,3]) &\Rightarrow \langle \tilde{}3, \mathsf{un}\rangle \\ \#(\mathsf{map}\ +\ [\langle 1,2\rangle, \langle 3,4\rangle, \langle 5,6\rangle]) &\Rightarrow \langle \tilde{}3, \mathsf{un}\rangle \\ \#(\mathsf{fold}\ +\ 0\ [1,2,3]) &\Rightarrow \mathsf{un}\ .\end{aligned}$$

Note also the following example

$$\#(\mathsf{map}\ /\ [\langle 1,0\rangle,\langle 2,0\rangle,\langle 3,0\rangle]) \Rightarrow \langle \tilde{}3,\mathsf{un}\rangle\ .$$

Examples of shape analysis of terms involving zip and append are given below.

$$\begin{array}{ll} \#(\mathsf{zip\ mat1\ mat2}) & \Rightarrow\ \mathsf{err} \\ \#(\mathsf{append}\ [1,2,3]\ [4,5,6]) & \Rightarrow\ \langle \tilde{}6,\mathsf{un}\rangle \\ \#(\mathsf{append}\ [[1,1],[2,2],[3,3]]\ [[1],[2],[3]]) & \Rightarrow\ \mathsf{err}\ . \end{array}$$

Shape analysis of the last example reveals a shape error since the two arguments have entries of different lengths.

4.2.4 Linear algebra

We conclude this section with shape analysis of the linear algebra operations defined in Section 3.3.

The shape of transpose is operationally equivalent to

$$\#\mathsf{transpose} \approx \lambda x.\langle \mathsf{fst}\ (\mathsf{snd}\ x), \langle \mathsf{fst}\ x, \mathsf{snd}\ (\mathsf{snd}\ x)\rangle\rangle\ .$$

So the shape of transpose simply swaps the two sizes representing the sizes of a matrix. Similarly

$$\begin{array}{rcl} \#\mathsf{multiply} & \approx & \lambda x,y.\mathsf{ifs\ eq}\ (\mathsf{fst}\ (\mathsf{snd}\ x))\ (\mathsf{fst}\ y) \\ & & \quad \mathsf{then}\ \langle \mathsf{fst}\ x, \langle \mathsf{fst}\ (\mathsf{snd}\ y), \mathsf{un}\rangle\rangle \\ & & \quad \mathsf{else\ err}\ . \end{array}$$

Shape analysis of matrix multiplication gives the expected results, as in

$$\begin{array}{l} \#(\mathsf{multiply\ mat1\ mat2})\ \Rightarrow\ \langle \tilde{}2, \langle \tilde{}1, \mathsf{un}\rangle\rangle \\ \#(\mathsf{multiply\ mat2\ mat1})\ \Rightarrow\ \mathsf{err}\ . \end{array}$$

4.3 Properties of shape analysis

In this section we prove several important properties of shape analysis, the most significant of which are the following two claims:

- Whenever the evaluation of a term results in a shape error, then its shape analysis reveals a shape error (Corollary 4.3.5).
- Whenever the evaluation of a term terminates, then so does that of its shape (Corollary 4.3.7).

Thus the first claim asserts that shape analysis does, indeed, reveal all shape errors in a term, while the second claim is necessary for regarding shape analysis as a viable kind of static program analysis. These two results are corollaries of Theorem 4.3.4 which we will prove shortly, but first we present three simple lemmas. The first of these shows that the eq combinator behaves the way we expect:

Lemma 4.3.1 If $\mathsf{eq}\ t\ t' \Rightarrow \mathsf{zero}$, then $t = t'$.

Proof. By induction on the evaluation complexity of $\mathsf{eq}\ t\ t'$. Then by case analysis on the last step of the evaluation. There are only four possible cases:

1.
$$\frac{\dfrac{t \Rightarrow \mathsf{un}}{\mathsf{eq}\ t \Rightarrow \mathsf{eq}\ \mathsf{un}} \quad t' \Rightarrow \mathsf{un}}{\mathsf{eq}\ t\ t' \Rightarrow \mathsf{zero}}$$

   Therefore $t = t'$.

2.
$$\frac{\dfrac{t \Rightarrow \mathsf{zero}}{\mathsf{eq}\ t \Rightarrow \mathsf{eq}\ \mathsf{zero}} \quad t' \Rightarrow \mathsf{zero}}{\mathsf{eq}\ t\ t' \Rightarrow \mathsf{zero}}$$

   Therefore $t = t'$.

3.
$$\frac{\dfrac{t \Rightarrow \mathsf{succ}\ u}{\mathsf{eq}\ t \Rightarrow \mathsf{eq}\ (\mathsf{succ}\ u)} \quad t' \Rightarrow \mathsf{succ}\ v \quad \mathsf{eq}\ u\ v \Rightarrow \mathsf{zero}}{\mathsf{eq}\ t\ t' \Rightarrow \mathsf{zero}}$$

   By applying the induction hypothesis to $\mathsf{eq}\ u\ v$ we get $u = v$ (actually $u \equiv v$ since they are values) and $\mathsf{succ}\ u \equiv \mathsf{succ}\ v$. Therefore $t = t'$.

4.

$$\frac{\dfrac{t \Rightarrow \langle u_1, u_2 \rangle}{\mathsf{eq}\ t \Rightarrow \mathsf{eq}\ \langle u_1, u_2 \rangle} \quad t' \Rightarrow \langle v_1, v_2 \rangle \quad \mathsf{and}\ (\mathsf{eq}\ u_1\ v_1)\ (\mathsf{eq}\ u_2\ v_2) \Rightarrow \mathsf{zero}}{\mathsf{eq}\ t\ t' \Rightarrow \mathsf{zero}}$$

where $\mathsf{and} \equiv \lambda x.\lambda y.\mathsf{ifs}\ x\ \mathsf{then}\ y\ \mathsf{else}\ x$. By inspecting and we can see that if it is to evaluate to zero, then both its arguments have to evaluate to zero. So $\mathsf{eq}\ u_1\ v_1 \Rightarrow \mathsf{zero}$ and $\mathsf{eq}\ u_2\ v_2 \Rightarrow \mathsf{zero}$. By the induction hypothesis, $u_1 \equiv v_1$ and $u_2 \equiv v_2$ and therefore $t = t'$.

∎

The next lemma describes the behaviour of err:

Lemma 4.3.2 $t = \mathsf{err}$ iff $t \Rightarrow \mathsf{err}$.

Proof. The "if" direction is immediate. For the reverse direction, let us consider, for example, the terms t and err in the context $(\lambda x.\mathsf{un})\ \Box$ to see that $t \Rightarrow \mathsf{err}$ whenever $t = \mathsf{err}$. ∎

The following lemma, the last one before the proof of Theorem 4.3.4, shows that # is well-behaved with respect to substitution.

Lemma 4.3.3 $\#(t[t'/x]) \equiv \#t[\#t'/x]$.

Proof. By induction on the structure of t. ∎

The following theorem shows that # respects evaluation, with the possible exception of detecting extra errors.

Theorem 4.3.4 *If $t \Rightarrow r$, then $\#t = \#r$ or $\#t \Rightarrow \mathsf{err}$.*

Proof. By induction on the evaluation complexity of t. Then by case analysis on the last step in the evaluation. We will present only the more complicated of these cases, the result for the rest is proved similarly. Let us suppose that $\#t \not\Rightarrow \mathsf{err}$ (and thus, by Lemma 4.3.2, $\#t \neq \mathsf{err}$). We then also get that $\#t \Rightarrow v$ implies $\#t = v$, by Lemma 3.2.5, and the proof becomes mainly a matter of equational reasoning.

1. If t is a value, then the result follows immediately.

2. Suppose $t \equiv t_1\ t_2$ and the last step in the evaluation of t was

$$\frac{t_1 \Rightarrow \lambda x.t_1' \quad t_2 \Rightarrow v' \quad t_1'[v'/x] \Rightarrow v}{t_1\ t_2 \Rightarrow v}$$

   Now $\#t \equiv (\#t_1)\ (\#t_2)$. By applying the induction hypothesis to t_1 and t_2, we get $\#t_1 = \#(\lambda x.t_1') \equiv \lambda x.\#t_1'$ and $\#t_2 = \#v'$.

$$\begin{array}{rcl} (\#t_1)\ (\#t_2) & = & (\lambda x.\#t_1')\ \#v' \\ & = & \#t_1'[v''/x] \ \text{ where } \#v' \Rightarrow v'' \\ & = & \#t_1'[\#v'/x] \\ & \equiv & \#(t_1'[v'/x]) \ \text{ by Lemma 4.3.3} \\ & = & \#v \ \text{ by the induction hypothesis .} \end{array}$$

3.

$$\frac{t_1 \Rightarrow \mathsf{fst} \quad t_2 \Rightarrow \langle v, v_2 \rangle}{t_1\ t_2 \Rightarrow v}$$

$$\begin{array}{rcl} \#t \equiv (\#t_1)\ (\#t_2) & = & \mathsf{fst}\ \langle \#v, \#v_2 \rangle \ \text{ by the induction hypothesis} \\ & = & \mathsf{fst}\ \langle v', v_2' \rangle \ \text{ where } \#v \Rightarrow v',\ \#v_2 \Rightarrow v_2' \\ & = & v' \\ & = & \#v\ . \end{array}$$

   Case $t_1 \Rightarrow \mathsf{snd}$ is proved analogously.

4.

$$\frac{t_1 \Rightarrow \mathsf{tl} \quad t_2 \Rightarrow [v]}{t_1\ t_2 \Rightarrow \mathsf{err}}$$

$$\begin{array}{rcl} \#t & = & \#\mathsf{tl}\ (\#\mathsf{sing}\ \#v) \ \text{ by the induction hypothesis} \\ & \equiv & (\lambda x.t')\ ((\lambda x.\langle \tilde{}1, x \rangle)\ \#v) \\ & & \quad \text{where } t' \equiv \mathsf{ifs\ eq}\ x\ (\mathsf{snd}\ y)\ \mathsf{then}\ \langle \mathsf{succ}\ (\mathsf{fst}\ y), \mathsf{snd}\ y \rangle\ \mathsf{else\ err} \\ & = & (\lambda x.t')\ ((\lambda x.\langle \tilde{}1, x \rangle)\ v') \ \text{ where } \#v \Rightarrow v' \\ & = & t'[\langle \tilde{}1, v' \rangle / x] \\ & = & \mathsf{err} \\ & \equiv & \#v \end{array}$$

5.

$$\frac{t_1 \Rightarrow \mathsf{length} \quad t_2 \Rightarrow v_1 :: v_2 \quad \mathsf{length}\ v_2 \Rightarrow m}{t_1\ t_2 \Rightarrow \mathsf{succ}\ m}$$

$$\begin{aligned} \#t &= \mathsf{fst}\ ((\lambda x, y.t')\ \#v_1\ \#v_2) \\ &\quad \text{where } t' \equiv \mathsf{ifs\ eq}\ x\ (\mathsf{snd}\ y)\ \mathsf{then}\ \langle \mathsf{succ}\ (\mathsf{fst}\ y), \mathsf{snd}\ y\rangle\ \mathsf{else\ err} \\ &= ((\lambda x, y.t')\ v_1'\ v_2') \\ &\quad \text{where } \#v_1 \Rightarrow v_1',\ \#v_2 \Rightarrow v_2' \\ &= \mathsf{fst}\ \langle \mathsf{succ}\ (\mathsf{fst}\ v_2'), \mathsf{snd}\ v_2'\rangle \\ &= \mathsf{succ}\ (\mathsf{fst}\ v_2')\ . \end{aligned}$$

Since, by the induction hypothesis,

$$\#(\mathsf{length}\ v_2) \equiv \mathsf{fst}\ \#v_2 = \#m\ ,$$

we get

$$\begin{aligned} \#t &= \mathsf{succ}\ \#n \\ &\equiv \#(\mathsf{succ}\ n)\ . \end{aligned}$$

6.

$$\frac{t'[\mathsf{rec}\ f.t'/f] \Rightarrow v}{\mathsf{rec}\ f.t' \Rightarrow v}$$

$$\begin{aligned} \#t \equiv \mathsf{rec}\ f.\#t' &= \#t'[\mathsf{rec}\ f.\#t'/f] \\ &\equiv \#(t'[\mathsf{rec}\ f.t'/f]) \\ &= \#v \quad \text{by the induction hypothesis}\ . \end{aligned}$$

Other cases are proved similarly. ∎

We immediately get the following corollary which shows that all shape errors are detected by shape analysis.

Corollary 4.3.5 *If* $t \Rightarrow \mathsf{err}$*, then* $\#t \Rightarrow \mathsf{err}$.

Proof. Follows immediately from Theorem 4.3.4 since the only value equal to the error is the error itself. ∎

Remarks

1. The previous corollary is probably the single most important result in this thesis, as it justifies our claim of being able to detect all shape errors in a term.

2. The converse of the previous corollary does not hold. As a counterexample, consider the vector $[[1], [1, 1]]$. This term is a value, but its shape analysis reveals a shape error, since the two entries have different lengths.

To prove the next corollary, we need this simple lemma.

Lemma 4.3.6 Let r be a result, then the evaluation of $\#r$ terminates.

Proof. By induction on the structure of r. ∎

The following corollary shows that the shape analysis of t terminates whenever the evaluation of t does.

Corollary 4.3.7 *If the evaluation of t terminates, then so does that of $\#t$.*

Proof. Suppose the evaluation of t terminates, i.e. $t \Rightarrow r$ for some result r. Theorem 4.3.4 says that either $\#t = \#r$ or $\#t \Rightarrow \mathsf{err}$. In the latter case the evaluation of $\#t$ obviously terminates. In the former case we can use Lemma 4.3.6 to see that the evaluation of $\#r$ terminates and therefore so does that of $\#t$ (by soundness). ∎

Again, the converse does not hold, the reason being that shape of a term may evaluate to err even when the term itself does not – this means that certain subterms in the shape may not be evaluated. The term

$$\langle [\tilde{}1, \tilde{}2], \mathsf{rec}\ f.f \rangle$$

is an example of a term whose shape analysis terminates, even though its evaluation does not. We also want to allow for the possibility of introducing other datum operations whose evaluation may not terminate but whose shape analysis does (shape analysis of such operations would typically be very easy as their shapes would be trivial).

Chapter 5

Simplifying shapes

As we have seen in Section 4.2, the shapes of higher-order terms, as produced by shape analysis, are often much more complicated than we would like them to be and than is strictly necessary. For example, the shape of matrix multiplication, as defined in Section 3.3, would not probably fit on one page while there exists a much simpler, operationally equivalent shape, namely

$$\begin{aligned}\lambda x, y.&\mathsf{ifs}\ \mathsf{eq}\ (\mathsf{fst}\ (\mathsf{snd}\ x))\ (\mathsf{fst}\ y)\\ &\mathsf{then}\ \langle \mathsf{fst}\ x, \langle \mathsf{fst}\ (\mathsf{snd}\ y), \mathsf{un}\rangle\rangle\\ &\mathsf{else}\ \mathsf{err}\ .\end{aligned}$$

Even though the two shapes give us the same assurances with regard to error-checking, it would often be useful to get the simpler of the two as the result of shape analysis. In this chapter we present a system capable of producing simple shapes for (a subsystem of) VEC without using any elaborate theorem proving techniques. Our hope is that this system, and the methods used, can be extended to cover a larger part, or possibly all of VEC. With this in mind, the chapter should be viewed as a basis for a possible way forward rather than a definitive piece of work.

In the first section we introduce SIZE_C, an extension of the SIZE language, as well as a reduction $\rightarrow_{\mathsf{sc}}$ on SIZE_C intended as a mechanism for simplifying SIZE_C terms. The following section then introduces $\#_\text{C}$, an alternative shape analysis of VEC. We will see that $\rightarrow_{\mathsf{sc}}$ is able to considerably simplify shapes of many VEC terms.

5.1 SIZE with checks

We introduce a modified version of the SIZE language called SIZE_C (standing for SIZE with checks). Its type system is the same as that of SIZE, that is given by

$$\sigma ::= \mathsf{un} \mid \mathsf{sz} \mid \sigma \times \sigma \mid \sigma \to \sigma \ .$$

As before, θ will represent the function-free types.

SIZE_C terms are given by

$$t \ ::= \ x \mid c \mid \lambda x^{\sigma}.t \mid t\ t \mid \mathsf{let}\ x \Leftarrow t\ \mathsf{in}\ t\ .$$

where c is a combinator, the set of which is shown in Fig 5.1. In particular, SIZE_C has neither conditionals nor recursion. To partially compensate for this omission, the set of combinators is augmented by the and and not combinators. We also introduce the combinator $\tilde{+}$ representing the operation of addition on sizes (of course, other algebraic operations on sizes can be introduced as well, but in the following we will need only addition).

un	:	un
$\mathsf{pair}^{\sigma,\sigma'}$	:	$\sigma \to \sigma' \to \sigma \times \sigma'$
$\mathsf{fst}^{\sigma,\sigma'}$	:	$\sigma \times \sigma' \to \sigma$
$\mathsf{snd}^{\sigma,\sigma'}$	:	$\sigma \times \sigma' \to \sigma'$
zero	:	sz
succ	:	$\mathsf{sz} \to \mathsf{sz}$
pred	:	$\mathsf{sz} \to \mathsf{sz}$
eq^{θ}	:	$\theta \to \theta \to \mathsf{sz}$
and	:	$\mathsf{sz} \to \mathsf{sz} \to \mathsf{sz}$
not	:	$\mathsf{sz} \to \mathsf{sz}$
$\tilde{+}$	:	$\mathsf{sz} \to \mathsf{sz} \to \mathsf{sz}$
err^{σ}	:	σ

Figure 5.1: SIZE_C combinators and their types

The main new feature in SIZE_C is the let constructor. Its type is as usual,

given by the inference rule below

$$\textbf{Let}\ \frac{\Gamma \vdash t' : \sigma' \quad \Gamma, x : \sigma' \vdash t : \sigma}{\Gamma \vdash \mathsf{let}\ x \Leftarrow t'\ \mathsf{in}\ t : \sigma}$$

Its semantics will be standard – when evaluating $\mathsf{let}\ x \Leftarrow t'\ \mathsf{in}\ t$ first evaluate t' to some v, substitute the result for x into t and then evaluate $t[v/x]$.

We may contract successive let's as below:

$$\mathsf{let}\ x_1, \ldots, x_n = t_1, \ldots, t_n\ \mathsf{in}\ t \ \text{ stands for } \ \mathsf{let}\ x_1 \Leftarrow t_1\ \mathsf{in}\ (\ldots \mathsf{let}\ x_n \Leftarrow t_n\ \mathsf{in}\ t)\ .$$

The main reason for introducing the let constructor is to allow us to build "checks for errors" into $\text{S}{\small IZE}_\text{C}$ terms. We therefore introduce the following syntactic sugar:

$$\mathsf{check}\ t'\ \mathsf{then}\ t \quad \equiv \quad \mathsf{let}\ x \Leftarrow t'\ \mathsf{in}\ t, \quad x \notin FV(t)$$

Thus when evaluating a term $\mathsf{check}\ t'\ \mathsf{then}\ t$ one first evaluates t', checks for errors and then evaluates t. check's will thus serve to keep track of those subterms which may evaluate to an error.

We introduce the following notation to improve readability:

$$\mathsf{check}\ t_1, \ldots, t_n\ \mathsf{then}\ t \ \text{ stands for } \ \mathsf{check}\ t_1\ \mathsf{then}\ (\ldots \mathsf{check}\ t_n\ \mathsf{then}\ t)$$
$$t_1 = t_2 \ \text{ stands for } \ \mathsf{pred}\ (\mathsf{not}\ (\mathsf{eq}\ t_1\ t_2))\ .$$

The latter notation will often be used in combination with check's to test for equality of two terms as $t_1 = t_2$ will reduce to err whenever t_1 and t_2 are not equal. The term $\mathsf{check}\ t_1 = t_2\ \mathsf{then}\ t$ thus means "return the value of t if t_1 and t_2 are equal and an error otherwise".

Definition 5.1.1 A *safe term* s is a term given by the following grammar

$$\begin{aligned} s ::= {} & x \mid c\ (\not\equiv \mathsf{err}) \mid \lambda x^\sigma.t \mid \mathsf{pair}\ s \mid \ \mathsf{pair}\ s\ s \mid \mathsf{fst}\ s \mid \mathsf{snd}\ s \\ & \mathsf{succ}\ s \mid \mathsf{eq}\ s \mid \mathsf{eq}\ s\ s \mid \mathsf{and}\ s \mid \mathsf{and}\ s\ s \mid \mathsf{not}\ s\ . \end{aligned}$$

Remarks

1. We shall use s to denote safe terms. Terms which are not safe will be denoted ns.

2. Safe terms can never reduce to err, not even under any substitution of other safe terms for their free variables. This means that we can safely β-reduce applications with safe terms as arguments as then there is no risk of losing an error.

The reduction rules for SIZE_C are generated by the base reductions $\rightarrow_{\mathsf{sc}}$ given in Fig 5.2 and 5.3. Most of the rules (those for applications and concerning let) are analogous to those of Moggi's λ_c calculus (see [Moggi, 1989]).

$$
\begin{array}{lll}
ns\ t & \rightarrow_{\mathsf{sc}} & \mathsf{let}\ x \Leftarrow ns\ \mathsf{in}\ x\ t\ , \quad x \text{ fresh} \\
s\ ns & \rightarrow_{\mathsf{sc}} & \mathsf{let}\ x \Leftarrow ns\ \mathsf{in}\ s\ x\ , \quad x \text{ fresh} \\
(\lambda x.\ t)\ s & \rightarrow_{\mathsf{sc}} & t[s/x] \\
\mathsf{fst}\ \langle t_1, t_2 \rangle & \rightarrow_{\mathsf{sc}} & \mathsf{check}\ t_2\ \mathsf{then}\ t_1 \\
\mathsf{snd}\ \langle t_1, t_2 \rangle & \rightarrow_{\mathsf{sc}} & \mathsf{check}\ t_1\ \mathsf{then}\ t_2 \\
\mathsf{pred}\ (\mathsf{succ}\ t) & \rightarrow_{\mathsf{sc}} & t \\
\mathsf{pred}\ \mathsf{zero} & \rightarrow_{\mathsf{sc}} & \mathsf{err} \\
\mathsf{let}\ x \Leftarrow t\ \mathsf{in}\ x & \rightarrow_{\mathsf{sc}} & t \\
\mathsf{let}\ x \Leftarrow s\ \mathsf{in}\ t & \rightarrow_{\mathsf{sc}} & t[s/x] \\
\mathsf{let}\ x \Leftarrow \mathsf{err}\ \mathsf{in}\ t & \rightarrow_{\mathsf{sc}} & \mathsf{err} \\
\mathsf{let}\ x \Leftarrow t\ \mathsf{in}\ \mathsf{err} & \rightarrow_{\mathsf{sc}} & \mathsf{err} \\
\mathsf{let}\ x \Leftarrow (\mathsf{let}\ y \Leftarrow t'\ \mathsf{in}\ t_1)\ \mathsf{in}\ t_2 & \rightarrow_{\mathsf{sc}} & \mathsf{let}\ y, x \Leftarrow t', t_1\ \mathsf{in}\ t_2
\end{array}
$$

Figure 5.2: Base reductions for SIZE_C

The (mutually exclusive) rules at the top of the figure describe the reduction of a function application. The algorithm for deciding which rule applies is simple. We first reduce the function to its normal form. If it does not reduces to a value, we use the first rule. If it does reduce to a value, we reduce the argument. If the argument does not reduce to a value, we use the second rule. Finally, if both the function and the argument reduce to values, either they together form a value or one of the following rules applies (depending on the value of the function).

Some of the other rules are modified so as to keep track of (potential) errors. Thus terms which would have been discarded in, say, the SIZE reduc-

and zero t	$\to_{\mathsf{sc}}$ t
and t zero	$\to_{\mathsf{sc}}$ t
and (succ t_1) t_2	$\to_{\mathsf{sc}}$ check t_1, t_2 then ~1
and t_1 (succ t_2)	$\to_{\mathsf{sc}}$ check t_1, t_2 then ~1
not zero	$\to_{\mathsf{sc}}$ ~1
not (succ t)	$\to_{\mathsf{sc}}$ check t then zero
eq zero t	$\to_{\mathsf{sc}}$ t
eq t zero	$\to_{\mathsf{sc}}$ t
eq (succ t_1) (succ t_2)	$\to_{\mathsf{sc}}$ eq t_1 t_2
eq un t	$\to_{\mathsf{sc}}$ check t then zero
eq t un	$\to_{\mathsf{sc}}$ check t then zero
eq t t	$\to_{\mathsf{sc}}$ check t then zero
eq $\langle t_1, t_2\rangle$ $\langle t'_1, t'_2\rangle$	$\to_{\mathsf{sc}}$ and (eq t_1 t'_1) (eq t_2 t'_2)

Figure 5.3: Equality reductions in SIZE$_{\text{C}}$

tion system (such as t_2 in fst $\langle t_1, t_2\rangle$) are kept as checks. Those checks which are guaranteed not to reduce to an error can then be eliminated by the rule

$$\mathsf{let}\ x \Leftarrow s\ \mathsf{in}\ t \ \to_{\mathsf{sc}}\ t[s/x]$$

This means that only checks of the form pred t_1 or f t_1, where f is a variable, cannot be eliminated. check's can be propagated towards the head of a term, though not across a λ. Similarly, terms containing errors reduce to an error (unless the error is under a λ) – we have contemplated introducing such rules already in Section 2.2.8. The rules in Fig 5.3 are mostly familiar (with some added checks to ensure no potential errors are lost) with the following exception:

$$\mathsf{eq}\ t\ t \to_{\mathsf{sc}} \mathsf{check}\ t\ \mathsf{then\ zero}$$

This rule is not strictly necessary but it is useful as it allows us to simplify terms such as eq x x (terms like this arise quite often in the shapes of VEC terms, see the following section). $\to_{\mathsf{sz}}$ can considerably simplify some SIZE$_{\text{C}}$ terms as we will see in the following section when we apply it to a number of examples.

The standard reduction lemmas about $\to_{\mathsf{sz}}$ (such as subject reduction)

can be proved but, as their proofs are similar to those for SIZE and VEC, we omit them.

Theorem 5.1.2 $\rightarrow$ *is confluent and strongly normalising.*

Proof. Analogous to the proof of those properties for the λ_c calculus (see [Moggi, 1988] and [Moggi, 1989]). ∎

The operational semantics $\Rightarrow_{\mathsf{sc}}$ of SIZE_C is an extension of that of SIZE. Semantics of the new constructs is given in Fig 5.4 – semantics of the rest is the same as in SIZE and therefore omitted. Operational equivalence $\approx_{\text{SC}}$ is also defined in the standard way (see e.g. Definition 2.1.36). The usual suite of lemmas listing $\Rightarrow_{\mathsf{sc}}$ properties, similar to that presented towards the end of Chapter 2, can then be proved, but we omit it. Note that, given the absence of general recursion, every SIZE_C term has a value (which is, of course, unique).

$$\frac{t' \Rightarrow_{\mathsf{sc}} v' \quad t[v'/x] \Rightarrow_{\mathsf{sc}} r}{\mathsf{let}\ x \Leftarrow t'\ \mathsf{in}\ t \Rightarrow_{\mathsf{sc}} r} \qquad \frac{t' \Rightarrow_{\mathsf{sc}} \mathsf{err}}{\mathsf{let}\ x \Leftarrow t'\ \mathsf{in}\ t \Rightarrow_{\mathsf{sc}} \mathsf{err}}$$

$$\frac{t_1 \Rightarrow_{\mathsf{sc}} \mathsf{not} \quad t_2 \Rightarrow_{\mathsf{sc}} \mathsf{zero}}{t_1\ t_2 \Rightarrow_{\mathsf{sc}} \mathsf{succ\ zero}} \qquad \frac{t_1 \Rightarrow_{\mathsf{sc}} \mathsf{not} \quad t_2 \Rightarrow_{\mathsf{sc}} \mathsf{succ}\ v}{t_1\ t_2 \Rightarrow_{\mathsf{sc}} \mathsf{zero}}$$

$$\frac{t_1 \Rightarrow_{\mathsf{sc}} \mathsf{and} \quad t_2 \Rightarrow_{\mathsf{sc}} v}{t_1\ t_2 \Rightarrow_{\mathsf{sc}} \mathsf{and}\ v} \qquad \frac{t_1 \Rightarrow_{\mathsf{sc}} \mathsf{and\ zero} \quad t_2 \Rightarrow_{\mathsf{sc}} v}{t_1\ t_2 \Rightarrow_{\mathsf{sc}} v}$$

$$\frac{t_1 \Rightarrow_{\mathsf{sc}} \mathsf{and}\ (\mathsf{succ}\ v) \quad t_2 \Rightarrow_{\mathsf{sc}} v'}{t_1\ t_2 \Rightarrow_{\mathsf{sc}} \mathsf{succ\ zero}}$$

Figure 5.4: Operational semantics of SIZE_C

Analogously as in SIZE can soundness of the calculus be proved:

Theorem 5.1.3 (Soundness theorem). *Let t be a closed term. If $t =_{\mathsf{sc}} t'$ then $t \approx_{\mathsf{sc}} t'$.*

Proof. Analogous to the proof of the soundness theorem for SIZE. ∎

In the rest of this chapter, we shall omit the subscripts as in $\Rightarrow_{\mathsf{sc}}$ as it is unlikely to cause any confusion.

5.2 Shape analysis with checks

The reason we introduced SIZE_{C}, as we had SIZE before, is that SIZE_{C} will be the target language of (a variant of) shape analysis of VEC (we shall use $\#_{\text{C}}$ to denote this alternative shape analysis). As before, the main goal is for $\#_{\text{C}}$ to be "sound", that is to satisfy (the equivalent of) Theorem 4.3.4, which says

$$t \Rightarrow_{\mathsf{vc}} r \qquad \text{implies} \qquad \#_{\text{C}} t \approx \#_{\text{C}} r \;\text{ or }\; \#_{\text{C}} t \Rightarrow \mathsf{err}\ .$$

The definition of $\#_{\text{C}}$ is essentially the same as that of $\#$ (and its action on types is completely unchanged) with the following two exceptions. It is not difficult to see that due to the lack of the recursion constructor, SIZE_{C} cannot, in general, express the shapes of VEC terms involving recursion and we thus put

$$\#_{\text{C}}(\mathsf{rec}\ f.t) \equiv \mathsf{err}$$

(which amounts to saying that we cannot analyse terms involving recursion). Of course, there are cases when the shape of a term involving recursion is expressible in SIZE_{C}, but detecting such cases would typically require a detailed analysis of the term in question and, in particular, some form of theorem proving. Combining shape analysis with different analysing techniques in this way would certainly be an option, but one we will not pursue in this chapter.

We have more options when defining the shape of a conditional (for example, letting

$$\#_{\text{C}}(\mathsf{ifs}\ t\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2) \equiv \mathsf{check}\ \#t, \#t_1 = \#t_2\ \mathsf{then}\ \#t_1$$

would be a possibility) but, so as not to complicate matters, we opt for the same approach, that is we put

$$\#_{\text{C}}(\mathsf{ifs}\ t\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2) \equiv \mathsf{err}\ .$$

The definition of $\#_{\text{C}}$ is summarised in Fig 5.5.

$$
\begin{array}{lcl}
\#_C x^\sigma & \equiv & x^{\#_C \sigma} \\
\#_C \mathsf{sing} & \equiv & \lambda x.\langle \mathsf{succ\ zero}, x\rangle \\
\#_C \mathsf{cons} & \equiv & \lambda x, y.\mathsf{check}\ x = (\mathsf{snd}\ y) \\
 & & \quad \mathsf{then}\ \langle \mathsf{succ}\ (\mathsf{fst}\ y), \mathsf{snd}\ y\rangle \\
\#_C \mathsf{hd} & \equiv & \mathsf{snd} \\
\#_C \mathsf{tl} & \equiv & \lambda x.\mathsf{check\ pred}\ (\mathsf{pred}\ (\mathsf{fst}\ x)) \\
 & & \quad \mathsf{then}\ \langle \mathsf{pred}\ (\mathsf{fst}\ x), \mathsf{snd}\ x\rangle \\
\#_C \mathsf{length} & \equiv & \mathsf{fst} \\
\#_C c & \equiv & c \quad \text{for other combinators } c \\
\#_C(\lambda x^\sigma.t) & \equiv & \lambda x^{\#_C\sigma}.\#_C t \\
\#_C(t_1\ t_2) & \equiv & \#_C t_1\ \#_C t_2 \\
\#_C(\mathsf{rec}\ f^\sigma.t) & \equiv & \mathsf{err} \\
\#_C(\mathsf{ifs}\ t\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2) & \equiv & \mathsf{err}
\end{array}
$$

Figure 5.5: Shape analysis in SIZE_C

Unfortunately, with shape analysis defined in this way, the shapes of many VEC terms, such as those defined in Section 3.3, will be errors since they typically involve recursion (and conditionals). Such a system would not be very useful, and we therefore extend VEC with additional vector combinators, such as

$$
\begin{array}{lcl}
\mathsf{map} & : & (\tau \to \tau') \to \mathsf{vec}\ \tau \to \mathsf{vec}\ \tau' \\
\mathsf{fold} & : & (\tau \times \tau' \to \tau') \to \tau' \to \mathsf{vec}\ \tau \to \tau' \\
\mathsf{zip} & : & \mathsf{vec}\ \tau \to \mathsf{vec}\ \tau' \to \mathsf{vec}\ (\tau \times \tau') \\
\mathsf{append} & : & \mathsf{vec}\ \tau \to \mathsf{vec}\ \tau \to \mathsf{vec}\ \tau \\
\mathsf{transpose} & : & \mathsf{mat}\ \tau \to \mathsf{mat}\ \tau\ .
\end{array}
$$

Programming language of this style is very similar to skeleton-based languages, (see [Darlington et al., 1993] or [Skillicorn, 1990]) and as such is well suited for parallelisation.

The shapes of these combinators are expressible in $\textsc{Size}_\mathrm{C}$ as follows:

$$
\begin{array}{lcl}
\#_\mathrm{C}\mathsf{map} & \equiv & \lambda f, x.\langle \mathsf{fst}\ x, f\ (\mathsf{snd}\ x)\rangle \\
\#_\mathrm{C}\mathsf{fold} & \equiv & \lambda f, x, y.\mathsf{check}\ (f\ (\mathsf{snd}\ y)) = x\ \mathsf{then}\ x \\
\#_\mathrm{C}\mathsf{zip} & \equiv & \lambda x, y.\mathsf{check}\ (\mathsf{fst}\ x) = (\mathsf{fst}\ y)\ \mathsf{then}\ \langle \mathsf{fst}\ x, \langle \mathsf{snd}\ x, \mathsf{snd}\ y\rangle\rangle \\
\#_\mathrm{C}\mathsf{append} & \equiv & \lambda x, y.\mathsf{check}\ (\mathsf{snd}\ x) = (\mathsf{snd}\ y)\ \mathsf{then}\ \langle (\mathsf{fst}\ x)\tilde{+}(\mathsf{fst}\ y), \mathsf{snd}\ x\rangle \\
\#_\mathrm{C}\mathsf{transpose} & \equiv & \lambda x.\langle \mathsf{fst}\ (\mathsf{snd}\ x), \langle \mathsf{fst}\ x, \mathsf{snd}\ (\mathsf{snd}\ x)\rangle\rangle\ .
\end{array}
$$

Of the shapes above, perhaps the only debatable one is that of fold – we check whether one application of the function we are folding doesn't change the shape of the result (and return an error otherwise). This is a safe approach though clearly it results in shape analysis returning an error more often than would be desirable – operations such as folding of cons will be analysed as ill-shaped. Folding of data-based operations, though, will not typically cause any problems.

This definition of $\#_\mathrm{C}$ satisfies the following equivalent of Theorem 4.3.4:

Theorem 5.2.1 *Let t be a closed* Vec *term. If $t \Rightarrow_\mathsf{vc} r$, then either $\#_\mathrm{C}t = \#_\mathrm{C}r$ or $\#_\mathrm{C}t \Rightarrow \mathsf{err}$.*

Proof. The proof is analogous to the proof of Theorem 4.3.4 and is therefore omitted. ∎

The previous theorem shows that we can safely replace shapes by their "simplified" versions. We will now demonstrate the simplifying power of $\rightarrow_\mathsf{sc}$ on the following examples.

We can express matrix multiplication in (the extended) Vec as follows:

$$
\begin{array}{lcl}
\mathsf{multiply} & : & \mathsf{mat\ nat} \rightarrow \mathsf{mat\ nat} \rightarrow \mathsf{mat\ nat} \\
\mathsf{multiply} & \equiv & \lambda x, y.\mathsf{map}\ (\lambda z.\mathsf{map}\ (\mathsf{scalar}\ z)\ (\mathsf{transpose}\ y))\ x
\end{array}
$$

where scalar' is the term $(\mathsf{fold}\ 0\ +) \circ (\mathsf{map}\ *) \circ \mathsf{zip}'$ (as before, prime denotes the uncurried version of a term). The shape of multiply is then a term in $\textsc{Size}_\mathrm{C}$, though still far from the simple shape we eventually want to get:

$$
\begin{array}{l}
\#_{\mathrm{C}}\mathsf{multiply} \equiv \lambda x, y.((\lambda f, x.\langle \mathsf{fst}\ x, f\ (\mathsf{snd}\ x)\rangle) \\
\quad (\lambda z.((\lambda f, x.\langle \mathsf{fst}\ x, f\ (\mathsf{snd}\ x)\rangle) \\
\quad\quad (((\lambda x, y.(\lambda z. \\
\quad\quad\quad (((\lambda f, y, z.\mathsf{check}\ (f\ (\mathsf{snd}\ z)) = y\ \mathsf{then}\ y) \\
\quad\quad\quad\quad (\lambda x.\mathsf{un}))\ \mathsf{un}) \\
\quad\quad\quad (\lambda y.((\lambda f, y.\langle \mathsf{fst}\ y, f\ (\mathsf{snd}\ y)\rangle)\ (\lambda x.\mathsf{un})) \\
\quad\quad\quad\quad ((\lambda x.((\lambda x, y. \\
\quad\quad\quad\quad\quad\quad \mathsf{check}\ \mathsf{fst}\ x = \mathsf{fst}\ y \\
\quad\quad\quad\quad\quad\quad \mathsf{then}\ \langle \mathsf{fst}\ x, \langle \mathsf{snd}\ x, \mathsf{snd}\ y\rangle\rangle) \\
\quad\quad\quad\quad\quad (\mathsf{fst}\ x)) \\
\quad\quad\quad\quad\quad (\mathsf{snd}\ x)) \\
\quad\quad\quad\quad y)) \\
\quad\quad\quad z)) \\
\quad\quad \langle x, y\rangle) \\
\quad\quad z)) \\
\quad\quad ((\lambda x.\langle \mathsf{fst}\ (\mathsf{snd}\ x), \langle \mathsf{fst}\ x, \mathsf{snd}\ (\mathsf{snd}\ x)\rangle\rangle)\ y))) \\
\quad x
\end{array}
$$

Fortunately, its normal form is much simpler:

$$
\begin{array}{rcl}
\#_{\mathrm{C}}\mathsf{multiply} & \to^* & \lambda x, y.\mathsf{check}\ \mathsf{fst}\ (\mathsf{snd}\ x) = \mathsf{fst}\ y \\
& & \quad\quad \mathsf{then}\ \langle \mathsf{fst}\ x, \langle \mathsf{fst}\ (\mathsf{snd}\ y), \mathsf{un}\rangle\rangle
\end{array}
$$

(you can make sure of this by using the implementation described in Appendix B).

Other matrix operations expressible in this system are, for example, the following operations on block matrices:

$$
\begin{array}{rcl}
\mathsf{putdown} & : & \mathsf{mat}\ \tau \to \mathsf{mat}\ \tau \to \mathsf{mat}\ \tau \\
\mathsf{putdown} & \equiv & \lambda x, y.\mathsf{append}\ x\ y
\end{array}
$$

and

$$
\begin{array}{rcl}
\mathsf{putright} & : & \mathsf{mat}\ \tau \to \mathsf{mat}\ \tau \to \mathsf{mat}\ \tau \\
\mathsf{putright} & \equiv & \lambda x, y.\mathsf{map}\ \mathsf{append}'\ (\mathsf{zip}\ x\ y)
\end{array}
$$

(where putdown places the second matrix below the first while putright places

it to the right). Their shapes then reduce the way we would expect:

$$\begin{array}{lll}
\#_{\mathrm{C}}\mathsf{putdown} & \rightarrow^* & \lambda x, y.\mathsf{check\ snd}\ x = \mathsf{snd}\ y \\
 & & \mathsf{then}\ \langle(\mathsf{fst}\ x)\tilde{}+(\mathsf{fst}\ y), \mathsf{snd}\ x\rangle \\
\#_{\mathrm{C}}\mathsf{putright} & \rightarrow^* & \lambda x, y.\mathsf{check\ fst}\ x = \mathsf{fst}\ y, \mathsf{snd}\ (\mathsf{snd}\ x) = \mathsf{snd}\ (\mathsf{snd}\ y) \\
 & & \mathsf{then}\ \langle \mathsf{fst}\ x, \langle(\mathsf{fst}\ (\mathsf{snd}\ x))\tilde{}+(\mathsf{fst}\ (\mathsf{snd}\ y)), \mathsf{snd}\ (\mathsf{snd}\ x)\rangle\rangle
\end{array}$$

Other examples of array-based operations and their shapes can be found in the implementation.

It turns out that we can simplify a significant proportion of shapes of array-based terms. Of course, the resulting shape (the normal form) is not guaranteed to be the simplest existing equivalent, and sometimes additional analyses can be employed to find a simpler, operationally equivalent term. These analyses might range from the simple (such as eliminating the superfluous check in $\mathsf{check}\ t', t'\ \mathsf{then}\ t$) to the sophisticated that use theorem-provers (such as eliminating the second check in $\mathsf{check}\ t' = \tilde{}1, t'\ \mathsf{then}\ t$). Clearly, the type of analysis one would choose depends both on the time (and the computing power) one is willing to spend, and on the urgency of getting as simple result as possible.

Chapter 6

Extending shape analysis

The Vec language, as presented in Chapter 3, was designed as a minimal language supporting vectors suitable for shape analysis. As such, it lacks many features commonly found in other functional languages and programming in Vec directly is therefore not as efficient as would be desirable. Even more importantly, some fundamental operations cannot even be expressed in it (we will talk about these shortly). In this chapter we will discuss possible extensions of Vec and investigate whether and how shape analysis can be extended to cover them and still retain the properties demonstrated in Chapter 4. Much of this chapter can be viewed as a discussion of the possible directions of future work.

The first section introduces the notion of a shapely operation and explains why these operations are fundamental to successful shape analysis. The second section then studies data-based conditionals and ways of adding them to the language. The third and fourth sections then examine the interaction between shape analysis and two different notions of polymorphism.

6.1 Shapely operations

We will now try to determine the criteria for identifying operations that can be added to Vec without compromising its shape analysis. In particular, we want to preserve the property stated in Theorem 4.3.4:

$$t \Rightarrow r \quad \text{implies} \quad \#t \cong \#r \text{ or } \#t \Rightarrow \mathsf{err}\ .$$

Moreover, we want to minimise the situations when the latter case arises – optimally, $\#t$ would evaluate to err only when there is a legitimate reason for regarding the term t as ill-shaped. Note that we can add *any* operation to VEC by letting its shape to be err – the above claim would then hold, but shape analysis of such operations would not produce any useful information, so this is not want we want. As we already discussed briefly in the Introduction, shape analysis can succeed only if the operations involved are *shapely*.

Definition 6.1.1 We say that an operation is *shapely* if the shape of its result is determined by the shape of its argument(s). The operation is *non-shapely* otherwise.

The previous definition is rather informal. There is, though, a related notion of a shapely natural transformation in the categorical semantics for shape, see [Jay, 1995].

VEC was designed so that all operations expressible in it were shapely (with the possible exception of revealing a shape error). Let us consider, for example, the operation append defined in Section 3.3 – the shape of the resulting vector, i.e. its length, can be determined once the lengths of the two input vectors are known:

$$\begin{aligned}\#(\mathsf{append}\ [1,2,3]\ [4,5]) &\cong \#([1,2,3])\ \tilde{}+\ \#([4,5])\\ &\cong \tilde{}\mathsf{3}\ \tilde{}+\ \tilde{}\mathsf{2}\\ &\cong \tilde{}\mathsf{5}\end{aligned}$$

where $\tilde{}+$ is the addition on sizes – this can already be defined in VEC using recursion.

On the other hand, filtering a vector of integers, say by the predicate > 0, is an example of a non-shapely operation – given only the length of the input vector we cannot determine the length of the result as this depends on the entries of the input, that is data. Had we added to VEC a combinator $\mathsf{filter} : (\mathsf{nat} \to \mathsf{bool}) \to \mathsf{vec\ nat} \to \mathsf{vec\ nat}$ with the usual reduction rules (though one has to be careful when dealing with singleton vectors)

$$\begin{array}{lll}\mathsf{filter}\ f\ [t] & \to_{\mathsf{vc}} [t] & \\ \mathsf{filter}\ f\ (t_1 :: t) & \to_{\mathsf{vc}} t_1 :: (\mathsf{filter}\ f\ t) & \text{when } f\ t_1\\ & \phantom{\to_{\mathsf{vc}}} \mathsf{filter}\ f\ t & \text{otherwise}\end{array}$$

(and equivalent operational semantics rules), we would not have been able to define $\#\mathsf{filter}$ such that we would have, for any $t : \mathsf{vec\ nat}$,

$$\mathsf{filter}\ t \Rightarrow v \quad \text{implies} \quad \#\mathsf{filter}\ \#t\ \cong\ \#v\ .$$

The only way to add filter to the language, and still preserve the equivalent of Theorem 4.3.4 would be to put

$$\#\mathsf{filter} \approx \lambda x, y.\mathsf{err} \quad \text{or} \quad \#\mathsf{filter} \approx \mathsf{err}$$

which would amount to admitting defeat as then basically any term involving filter would be regarded by shape analysis as ill-shaped.

We can now better understand some of the design decisions taken when VEC was introduced. The overriding concern was to ensure that only shapely operations were expressible in the language. The decision to introduce shape-based conditionals, that is conditionals branching according to a size, is a case in point, since conditionals branching according to a boolean (a datum) are, in general, non-shapely (we shall discuss such conditionals in the next section).

We have already mentioned, in Section 3.3, that we cannot introduce to the language the operation $\mathsf{nat_to_sz} : \mathsf{nat} \to \mathsf{sz}$ converting a natural number to the corresponding size. Since

$$\#(\mathsf{nat} \to \mathsf{sz}) \equiv \mathsf{un} \to \mathsf{sz}\ ,$$

the only shapely operations of type $\mathsf{nat} \to \mathsf{sz}$, apart from the ubiquitous err, are (those operationally equivalent to) either $\lambda x.\mathsf{err}$ or $\lambda x.\tilde{}\mathsf{n}$ for some $\tilde{}\mathsf{n} : \mathsf{sz}$, i.e. only the constant functions.

Of course, one can have a conversion going the other way $\mathsf{sz_to_nat} : \mathsf{sz} \to \mathsf{nat}$ with the reduction rules

$$\begin{array}{lll}\mathsf{sz_to_nat\ zero} & \to_{\mathsf{vc}} & 0 \\ \mathsf{sz_to_nat}\ (\mathsf{succ}\ n) & \to_{\mathsf{vc}} & 1 + (\mathsf{sz_to_nat}\ n)\ .\end{array}$$

and its shape given by

$$\#\mathsf{sz_to_nat} \equiv \lambda x.\mathsf{un}\ .$$

This operation can actually be already defined in VEC using the recursion operator.

We can see that there are operations, such as filter, which can never be successfully analysed. Fortunately, many array-based scientific computations of interest, such as almost all linear algebra operations, are, in fact, shapely. Moreover, shape analysis can still be of use even when a system allows the construction of non-shapely operations. In such cases it can be used as the basis for a "soft-shaping" analysis (analogous to soft typing), one which gives non-trivial results for some terms only, and would give the "can not be determined" answer for the rest (the simplest, though not necessarily the most useful, way to achieve this would be simply to let the shape of any non-shapely combinator or construct be err, as discussed above). Alternatively, shape analysis of the shapely parts can be combined with evaluation of the remainder. Of course, this latter approach is more suited to a dynamic analysis then a static one [Jay, 1996].

6.2 Data conditionals

One feature notably missing in VEC is a *data conditional*, i.e. a conditional branching according to a value of type nat (or bool or any other datum type). The only conditional constructor in VEC, ifs, branches according to a size, that is a shape, and as we discussed in the previous section, there is no (and there can not be any) nat to sz conversion operation, except a constant one. This means, for example, that the factorial function on natural numbers is not expressible in the language. The definition of the factorial would look something like

$$\mathsf{fact} \equiv \mathsf{rec}\ f^{\mathsf{nat}\to\mathsf{nat}}.\lambda x^{\mathsf{nat}}.\mathsf{ifs}\ x = 0\ \mathsf{then}\ 1\ \mathsf{else}\ x * (f\ \ (x-1))$$

but this expression is ill-typed – the type of the condition $x = 0$ could be either nat or bool, depending on the type of $=$, but it cannot be sz.

The inability to define functions such as the factorial on natural numbers is unfortunate, but the omission of data conditionals from the language is not an oversight on our part, as these are not, at least in their general form, shapely constructs. This can readily be seen from the following example.

Suppose we introduce such a term constructor, $\mathsf{if}^{\sigma}\ t\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2 : \sigma$

where $t : \mathsf{nat}$ and t_1 and t_2 are of type σ and the usual rules such as

$$\begin{array}{lll} \mathsf{if}\ 0\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2 & \to_{\mathsf{vc}}\ t_1 & \\ \mathsf{if}\ n\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2 & \to_{\mathsf{vc}}\ t_2 & \text{for numeral } n \not\equiv 0 \\ \mathsf{if}\ \mathsf{err}\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2 & \to_{\mathsf{vc}}\ \mathsf{err}\ . & \end{array}$$

What could the shape of this construct be? Clearly, since shape analysis should respect the typing (see Lemma 4.1.3), the type of the shape of the conditional has to be $\#\sigma$. Consider now the following two reductions:

$$\begin{array}{l} (\lambda x^{\mathsf{nat}}.\mathsf{if}\ x\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2)\ 0\ \to_{\mathsf{vc}}\ t_1 \\ (\lambda x^{\mathsf{nat}}.\mathsf{if}\ x\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2)\ 1\ \to_{\mathsf{vc}}\ t_2 \end{array}$$

But, regardless of what we define the shape of if to be, we get

$$\begin{array}{rcl} \#((\lambda x^{\mathsf{nat}}.\mathsf{if}\ x\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2)\ 0) & \equiv & (\lambda x^{\mathsf{un}}.\#(\mathsf{if}\ x\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2))\ \mathsf{un} \\ & \equiv & \#((\lambda x^{\mathsf{nat}}.\mathsf{if}\ x\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2)\ 1) \end{array}$$

and thus

$$\#t_1 \approx \#t_2\ .$$

In other words, if the two branches of a data conditional have different shapes, shape analysis can produce wrong shapes. This is not surprising, since the choice of the branch depends on a datum, and if the shapes of the branches differ, the construct is non-shapely. This means that shape analysis of a term such as

$$\mathsf{if}\ t\ \mathsf{then}\ [1,2,3]\ \mathsf{else}\ [1,2,3,4]$$

cannot always give the correct result. The most we can hope for is to force shape analysis to reveal a shape error in such cases. We now present three possible ways of adding data conditionals to Vec, each with its strengths and weaknesses.

The simplest approach would be to simply ignore the problem, and shift the responsibility for ensuring this equality to the programmer. We could then have data conditionals with branches of any type, and its shape could be given by

$$\#(\mathsf{if}\ t\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2) \equiv (\lambda x.\#t_1)\ \#t$$

(where x is a fresh variable, and the application is there only to "check" whether $\#t$ is not an error. We could equally well replace $\#t_1$ by $\#t_2$,

as this approach would give correct shapes only when $\#t_1$ and $\#t_2$ were operationally equal.) Of course, (the equivalent of) Theorem 4.3.4 would then have to be stated in terms of "well-shaped" terms only, that is terms with the two branches of any data conditional having the same shape.

A more constructive way would be to let shape analysis itself determine the equality of the shapes by letting, say,

$$\#(\mathsf{if}\ t\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2) \equiv (\lambda x.\mathsf{ifs}\ (\mathsf{eq}\ \#t_1\ \#t_2)\ \mathsf{then}\ \#t_1\ \mathsf{else}\ \mathsf{err})\ \#t$$

where, as above, x is fresh. This approach would mean that t_1 and t_2 had to be of a type whose shape supports equality, probably of ground type. However, the problem with this approach would be in the evaluation – were if lazy (as it probably would be, otherwise why not introduce it as a combinator), only one of t_1 and t_2 would be evaluated when evaluating $\mathsf{if}\ t\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2$, while both $\#t_1$ and $\#t_2$ would be evaluated when evaluating its shape. When considered together with recursion, this would mean that shape analysis of terms such as fact would not terminate even when the evaluation of the term itself did. Theorem 4.3.4 would have to be modified accordingly.

Another way to introduce data conditionals (and avoid non-terminating analyses) is to restrict the type of the recursion. We know, without unravelling the recursion, what the shape of a recursive function over, say, integers (such as the factorial) should be – such a function either always returns an error (and its shape is thus equivalent to either $\mathsf{err}^{\#(\mathsf{int}\to\mathsf{int})}$ or $\lambda x^{\#\mathsf{int}}.\mathsf{err}^{\#\mathsf{int}}$) or it returns a non-error integer. Since all such integers have the same shape, the shape of the function has to be (equivalent to) $\lambda x^{\#\mathsf{int}}.\mathsf{un}$. We can generalise this observation to tuples of datum types (in other words, ground discrete types ρ) – for each such type ρ, there is a canonical non-error shape (denoted $\mathsf{bang}^{\#\rho}$) which is just a tuple of un. The shapes of functions over such types ρ then have to be equivalent to one of $\mathsf{err}^{\#(\rho\to\rho)}$, $\lambda x^{\#\rho}.\mathsf{err}^{\#\rho}$ or $\lambda x^{\#\rho}.\mathsf{bang}^{\#\rho}$. If we restrict the recursion construction $\mathsf{rec}\ f^{\sigma}.t$ to types σ of the form $\rho \to \rho$ only, we can put

$$\#(\mathsf{rec}\ f^{\rho\to\rho}.t) \equiv \#t[\lambda x^{\#\rho}.\mathsf{bang}/f]\ .$$

Shape analysis would then retain the property to detect all shape errors, and would now always terminate (since shapes would not involve recursion). The factorial function would be expressible, but we would lose general recursion and with it much of expressibility of VEC. As a partial compensation, we could introduce new combinators, such as the iterator iter or some

second-order vector combinators such as map and fold. A similar approach to avoiding non-terminating analyses was adopted in [Jay et al., 1997].

6.3 Data polymorphism

VEC is a simply typed language – each term has exactly one type in a given context. However, a majority of functional (and, indeed, imperative) languages in use today support some kind of type polymorphism. The kind used most often is undoubtedly the Hindley-Milner style of polymorphism. First introduced by Milner in [Milner, 1978], building on the ideas from [Hindley, 1969], this kind of polymorphism allows us to dispense with type annotations in terms. The (raw) terms are generated by the following grammar

$$t \ ::= \ x \mid \lambda x.t \mid t\ t \mid \mathtt{let}\ x = t\ \mathtt{in}\ t\ .$$

One can then infer the types a term can have. Of course, a term such as $\lambda x.x$ can have many types, in fact any type of the form $\sigma \to \sigma$ would do. Similarly the type of $\lambda f.\lambda x.f(x)$ can be any type of the form $(\sigma \to \tau) \to \sigma \to \tau$. This suggests a way to describe the types of terms using *type variables*. One can then get any type a term can have by substituting types for these variables. The types system then may look like this

$$\sigma \ ::= \ \alpha \mid \sigma \to \sigma$$

where α is a type variable (usually another layer of *type schema* is introduced on top of the types σ, but we shall not go into the details as they are well known and documented).

This type system is at the core of many functional languages such as ML [Harper et al., 1986] or Haskell [Hudak et al., 1992], and it would be natural to ask whether we can extend VEC (and SIZE, naturally) with this kind of polymorphism. Indeed we can, and the approach would be quite standard. We would have to develop a type inference algorithm for such polymorphic VEC along the lines of the algorithm $\mathcal{W}$ ([Tofte, 1990]), and some of the results in Chapters 2 and 3 would have to be modified accordingly. The most important observation from our perspective is that shape analysis would be essentially unchanged, and the results corresponding to those from Chapter 4 would still hold. This suggests that (Hindley-Milner) polymorphism

is orthogonal to shape analysis and that was also the reason we decided to work with a simply-typed language – polymorphism does not bring any new insights to the problem and would only complicate the presentation.

6.4 Shape polymorphism

Hindley-Milner polymorphism, discussed briefly in the previous section, is only one, though widely used, of many polymorphic type systems. In fact, shape itself has led to the discovery of a new kind of polymorphism, one closely connected to shape analysis.

An example of a data polymorphic operation is map – regardless of the type of the list entries, the overall algorithm remains the same – go through the list and apply the function being mapped to each entry. The type of map typically is

$$\mathsf{map} : (\alpha \to \beta) \to \mathsf{list}\ \alpha \to \mathsf{list}\ \beta$$

with α and β ranging over types. It is easy to see that the same mechanism can be applied when mapping a function over other data structures as well – whether we are mapping a function over a matrix or a tree, the algorithm remains the same – go through all the data and apply the function to each. Thus other types of map might be

$$\begin{aligned}\mathsf{map} &: (\alpha \to \beta) \to \mathsf{mat}\ \alpha \to \mathsf{mat}\ \beta \\ \mathsf{map} &: (\alpha \to \beta) \to \mathsf{tree}\ \alpha \to \mathsf{tree}\ \beta\ .\end{aligned}$$

In general, the type of map might be any type of the form

$$\mathsf{map} : (\alpha \to \beta) \to F\ \alpha \to F\ \beta\ .$$

where F is a type constructor of a suitable kind. The question now arises whether it is possible to describe the type and/or action of map uniformly.

We have already mentioned some of the recent approaches to this and related issues (polytypism [Jeuring and Jansson, 1996], intensional polymorphism [Harper and Morrisett, 1995]). Shape theory suggests using shape polymorphic type systems [Jay and Cockett, 1994]. In such a type system, the type of map is parametric in the choice of the datatype constructor, covering thus all the cases mentioned above. ML has been extended with shape polymorphism in the FML language [Bellè et al., 1996].

Introducing shape polymorphism to VEC would allow us, for example, to work with empty vectors using the combinator nil. As we already discussed in Section 4.1, we have introduced the singleton as the elementary vector constructor in order to be able to check for the equality of (the shapes of) entries when consing additional elements, an ability essential for successful shape analysis. This would not be possible with the simply typed version of nil as the empty vector does not have any entries. On the other hand, had nil been introduced as a shape polymorphic combinator, it would have been able to represent the empty vector with entries of *any* shape, thus getting around the problem. We are confident that shape analysis can be combined with shape polymorphism in a single language, giving us both the safety of shape analysis and the power of shape polymorphism. This remains to be done in the future.

Chapter 7

Conclusions

Shape theory represents what we believe is the most general attempt to date to unify the various notions of shape (the shape of an array, the shape of a tree and so on) under a single framework. This uniform treatment of shapes has important implications for programming language design as it allows separate manipulations of shapes and data. Shape analysis, a branch of shape theory, exploits this shape/data split by using shapes as a basis for compile time program optimisations. These can range from speeding up the program's performance to improved memory allocation. This thesis focused on another application, using shape analysis for error detection.

Shape errors, that is errors arising from working with ill-formed or incompatible shapes, form an important class of program errors. Their examples include zipping together vectors of unequal lengths or multiplying matrices of ill-matched sizes. Significantly, array access errors, an intensely studied class of errors, can also be viewed in this light.

This thesis has concentrated on shape analysis (and error detection) of arrays and array based programs. The VEC language, a functional language based on the simply-typed lambda calculus supporting vectors, has been introduced as a vehicle for this study. To ensure that shape analysis is able to handle all VEC terms, several novel design decisions had to be made. These included introducing the type of sizes, a new kind of natural numbers used as vector lengths and treated as shapes.

Shapes of VEC terms have been isolated in its sublanguage SIZE. SIZE

terms can be regarded as pure shapes, as they do not involve any data or data-based computations.

Two kinds of shape analysis on VEC have been studied. The first one has been defined as a translation mapping a term in VEC terms to its shape in SIZE. The main technical result of the thesis then showed that shape analysis of a VEC term respects its evaluation (with the possible exception of detecting extra shape errors). Formally this can be expressed as

$$t \Rightarrow v \quad \text{implies} \quad \#t \approx \#v \text{ or } \#t \Rightarrow \mathsf{err}\ .$$

A corollary of the above result showed that shape analysis detects all shape errors. Another important result proved that shape analysis of a term terminates whenever its evaluation does, a result essential for regarding shape analysis as a static program analysis.

Later a modified form of shape analysis has been introduced, mapping VEC terms to their shapes in SIZE_C. The SIZE_C language, a variant of SIZE, supports a novel kind of simplifying reduction on terms. Thus shape analysis on SIZE_C, though suitable only for a restricted set of VEC terms, is able to produce very simple shapes of some higher order terms, including matrix multiplication. These shapes can then be used, for example, as a program verification tool.

Shape analysis similar to that presented here has already been used for several important applications. It has been the basis for estimating parallel execution costs of VEC programs in the PRAM setting as described in [Jay et al., 1997]. Work is now under way to implement a prototype shape-based Algol-like language, FISh (an acronym for Functional and Imperative SHape) [Jay and Steckler, 1998]. FISh uses shape analysis to generate efficient implementations (with execution speeds comparable to those of imperative languages) of programs written in a high-level programming style, similar to that supported by functional languages.

Shape analysis may in future be used both as an error detection and a program optimisation tool. For example, it can be used in compilers to estimate the sizes of the data structures involved and thus improve memory allocation – the FISh compiler uses shape analysis to avoid unnecessary boxing of data. The ability of shape analysis to detect array access errors can also be exploited in new optimising techniques.

Future work may extend shape analysis to other language constructs and thus richer languages. Some of these extensions (data-based conditionals, data and shape polymorphism) have already been discussed in this thesis. It is hoped that, ultimately, shape analysis will be combined with other shape-based features, such as shape polymorphism, in a single language which would be able to provide unprecedented flexibility and performance.

Bibliography

[Abramsky and Hankin (editors), 1987] Abramsky, S. and Hankin (editors), C. (1987). *Abstract Interpretation of Declarative Languages.* Series in Computers and Their Applications. Ellis Horwood.

[Aho et al., 1986] Aho, A., Sethi, R., and Ullman, J. (1986). *Compilers: Principles, Techniques and Tools.* Addison-Wesley.

[Asperti and Longo, 1991] Asperti, A. and Longo, G. (1991). *Categories, Types and Structures : An introduction to category theory for the working computer scientist.* Foundations of Computing Series. Massachusetts Institute of Technology.

[Asuru, 1992] Asuru, J. (1992). Optimization of array subscript range checks. *ACM Letters on Programming Languages and Systems*, 1(2):109–118.

[Barendregt, 1984] Barendregt, H. (1984). *The Lambda Calculus: Its Syntax and Semantics.* North Holland. revised edition.

[Barendregt, 1992] Barendregt, H. (1992). Lambda calculi with types. In Abramsky, S., Gabbay, D., and Maibaum, T., editors, *Handbook of Logic in Computer Science, volume 2*, pages 1–116. Oxford University Press.

[Barr and Wells, 1990] Barr, M. and Wells, C. (1990). *Category Theory for Computing Science.* International Series in Computer Science. Prentice Hall.

[Bellè et al., 1996] Bellè, G., Jay, C. B., and Moggi, E. (1996). Functorial ML. In *PLILP '96*, volume 1140 of *Lecture Notes in Computer Science*, pages 32–46. Springer Verlag.

[Bellè and Moggi, 1997] Bellè, G. and Moggi, E. (1997). Typed intermediate languages for shape-analysis. In *Proceedings of Rewriting Techinques and Applications 1997.* to appear.

[Blelloch, 1992] Blelloch, G. (1992). NESL: a nested data parallel language (version 3.1). Technical Report CMU-CS-95-170, School of Computer Science, Carnegie-Mellon University.

[Blelloch et al., 1991] Blelloch, G., Chatterjee, S., and Fisher, A. (1991). Size and access inference for data-parallel programs. In *ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 130–144.

[Ching, 1986] Ching, W. (1986). Program analysis and code generation in an APL/370 compiler. *IBM Journal of Research and Development*, 30(6):594–602.

[Church, 1940] Church, A. (1940). A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68.

[Cormen et al., 1990] Cormen, T., Leiserson, C., and R.L.Rivest (1990). *Introduction to Algorithms.* MIT Press, McGraw-Hill, New York.

[Cousot and Cousot, 1979] Cousot, P. and Cousot, R. (1979). Systematic design of program analysis frameworks. In *Proceedings of the Sixth ACM Annual Symposium on Programming Languages*, pages 269–282.

[Cousot and Halbwachs, 1978] Cousot, P. and Halbwachs, N. (1978). Automatic discovery of linear constraints among variables of a program. In *Conference Record of the 5th ACM Symposium on Principles of Programming Languages*, pages 84–97.

[Curry, 1934] Curry, H. (1934). Functionality in combinatory logic. *Proc. Nat. Acad. Science USA*, 20:584–590.

[Darlington et al., 1993] Darlington, J., Field, A., Harrison, P., Kelly, P., Wu, Q., and While, R. (1993). Parallel programming using skeleton functions. In *PARLE93, Parallel Architectures and Languages Europe: 5th International PARLE Conference*, pages 146–160.

[Dershowitz and Jouannaud, 1990] Dershowitz, N. and Jouannaud, J.-P. (1990). Rewrite systems. In van Leeuwen, J., editor, *Handbook of Theoretical Computer Science, volume B*. MIT Press.

[Duff et al., 1986] Duff, I., Erisman, A., and Reid, J. (1986). *Direct Methods for Sparse Matrices*. Clarendon Press Oxford.

[Feautrier, 1991] Feautrier, P. (1991). Dataflow analysis of scalar and array references. *International Journal of Parallel Programming*, 20(1):23–53.

[Ghani, 1995] Ghani, N. (1995). *Adjoint Rewriting*. PhD thesis, University of Edinburgh.

[Girard et al., 1989] Girard, J.-Y., Lafont, Y., and Taylor, P. (1989). *Proofs and Types*, volume 7 of *Tracts in Theoretical Computer Science*. Cambridge University Press.

[Gunter, 1992] Gunter, C. (1992). *Semantics of Programming Languages*. Foundations of Computing. MIT.

[Gupta, 1993] Gupta, R. (1993). Optimising array bound checks using flow analysis. *ACM Letters on Programming Languages and Systems*, 2:135–150.

[Gustavson et al., 1970] Gustavson, F., Liniger, W., and Willoughby, R. (1970). Symbolic generation of an optimal crout algorithm for sparse systems of linear equations. *Journal of the ACM*, 17(1):87–109.

[Harper et al., 1986] Harper, R., MacQueen, D., and Milner, R. (1986). Standard ML. Technical Report ECS-LFCS-86-2, Edinburgh Univ., Dept. of Comp. Sci.

[Harper and Morrisett, 1995] Harper, R. and Morrisett, G. (1995). Compiling polymorphism using intensional type analysis. In *Conference Record of POPL '95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 130–141, San Francisco, California.

[Harrison, 1977] Harrison, W. (1977). Compiler analysis of the value ranges for variables. *IEEE Transactions on Software Engineering*, 3(3):243–250.

[Hindley, 1969] Hindley, R. (1969). The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematics Society*, 146:29–60.

[Hindley and Seldin, 1986] Hindley, R. and Seldin, J. (1986). *Introduction to combinators and lambda calculus*. Cambridge University Press.

[Hoogendijk, 1997] Hoogendijk, P. (1997). *A generic theory of data types*. PhD thesis, Technische Universiteit Eindhoven.

[Hudak et al., 1992] Hudak, P., Peyton Jones, S., and Wadler, P. (1992). Report on the programming language Haskell: a non-strict, purely functional language, version 1.2. Technical report, University of Glasgow.

[Iverson, 1962] Iverson, K. (1962). *A Programming Language*. Wiley, New York, NY.

[Jay, 1994] Jay, C. (1994). Matrices, monads and the fast fourier transform. In *Proceedings of the Massey Functional Programming Workshop 1994*, pages 71–80.

[Jay, 1995] Jay, C. (1995). A semantics for shape. *Science of Computer Programming*, 25:251–283.

[Jay, 1996] Jay, C. (1996). Shape in computing. *ACM Computing Surveys*, 28(2):355–357.

[Jay and Cockett, 1994] Jay, C. and Cockett, J. (1994). Shapely types and shape polymorphism. In Sannella, D., editor, *Programming Languages and Systems - ESOP '94: 5th European Symposium on Programming, Edinburgh*, Lecture Notes in Computer Science, pages 302–316.

[Jay et al., 1997] Jay, C., Cole, M., Sekanina, M., and Steckler, P. (1997). A monadic calculus for parallel costing of a functional language of arrays. In Lengauer, C., Griebl, M., and Gorlatch, S., editors, *Euro-Par'97 Parallel Processing*, volume 1300 of *Lecture Notes in Computer Science*, pages 650–661. Springer.

[Jay and Ghani, 1995] Jay, C. and Ghani, N. (1995). The virtues of eta-expansion. *Journal of Functional Programming*, 5(2):135–154.

[Jay and Sekanina, 1997] Jay, C. and Sekanina, M. (1997). Shape checking of array programs. In Harland, J., editor, *Computing: the Australasian Theory Seminar, Proceedings, 1997*, volume 19 of *Australian Computer Science Communications*, pages 113–121.

[Jay and Steckler, 1998] Jay, C. and Steckler, P. (1998). The functional imperative: shape! In Hankin, C., editor, *Proceedings ESOP'98*. to appear.

[Jeuring and Jansson, 1996] Jeuring, J. and Jansson, P. (1996). Polytypic programming. In Launchbury, J., Meijer, E., and Sheard, T., editors, *Advanced Functional Programming, Second International School*, volume 1129, pages 68–114. Springer-Verlag. Lecture Notes in Computer Science.

[Jones et al., 1993] Jones, N., Gomard, C., and P.Sestoft, editors (1993). *Partial Evaluation and Automatic Program Generation*. International Series in Computer Science. Prentice-Hall.

[Jones and Nielson, 1992] Jones, N. and Nielson, F. (1992). Abstract interpretation: a semantics based tool for program analysis. In Abramsky, S., Gabbay, D., and Maibaum, T., editors, *Handbook of Logic in Computer Science, volume 4*, pages 527–636. Oxford University Press.

[Klop, 1980] Klop, J. (1980). *Combinatory Reduction Systems*. Number 127 in Mathematical Centre Tracts. Mathematisch Centrum, Amsterdam.

[Klop, 1992] Klop, J. (1992). Term rewriting systems. In Abramsky, S., Gabbay, D., and Maibaum, T., editors, *Handbook of Logic in Computer Science, v.2*. Oxford University Press.

[Kolte and Wolfe, 1995] Kolte, P. and Wolfe, M. (1995). Elimination of redundant array subscript range checks. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation, La Jolla, California*, pages 270–278.

[Kumar et al., 1994] Kumar, V., Grama, A., Gupta, A., and Karypis, G. (1994). *Introduction to parallel computing : design and analysis of algorithms*. Benjamin/Cummings Pub. Co., Redwood City, California.

[Lambek and Scott, 1986] Lambek, J. and Scott, P. (1986). *Introduction to Higher-Order Categorical Logic*, volume 7 of *Cambridge Studies in Advanced Mathematics*. Cambridge University Press.

[Markstein et al., 1982] Markstein, V., Cocke, J., and Markstein, P. (1982). Optimization of range checking. In *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction*, pages 114–119.

[Martin-Löf, 1984] Martin-Löf, P. (1984). *Intuitionistic Type Theory*. Bibliopolis.

[Meijer et al., 1991] Meijer, E., Fokkinga, M., and Paterson, R. (1991). Functional programming with bananas, lenses, envelopes and barbed wire. In *Proceedings of the 5th ACM Conference on Functional Programming and Computer Architecture*, volume 523 of *Lecture Notes in Computer Science*, pages 124–144. Springer Verlag.

[Milner, 1978] Milner, R. (1978). A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375.

[Mogensen, 1986] Mogensen, T. (1986). The application of partial evaluation to ray-tracing. Master's thesis, DIKU, University of Copenhagen, Denmark.

[Moggi, 1988] Moggi, E. (1988). *The Partial Lambda Calculus*. PhD thesis, University of Edinburgh.

[Moggi, 1989] Moggi, E. (1989). Computational lambda-calculus and monads. In *4th LICS Conf.* IEEE.

[Nielson and Nielson, 1992] Nielson, F. and Nielson, H. (1992). *Two Level Functional Languages*, volume 34 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press.

[Plotkin, 1975] Plotkin, G. (1975). Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1.

[Sagiv et al., 1996] Sagiv, M., Reps, T., and Wilhelm, R. (1996). Solving shape-analysis problems in languages with destructive updating. In *24th ACM Symposium on Principles of Programming Languages*, pages 16–31.

[Skillicorn, 1990] Skillicorn, D. (1990). Architecture-independent parallel computation. *IEEE Computer*, 23, No.12:38–51.

[Skillicorn, 1994] Skillicorn, D. (1994). *Foundations of Parallel Programming*. Cambridge Series in Parallel Computation 6. Cambridge University Press.

[Suciu and Tannen, 1994] Suciu, D. and Tannen, V. (1994). Efficient compilation of high-level data parallel algorithms. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*.

[Suzuki and Ishihata, 1977] Suzuki, N. and Ishihata, K. (1977). Implementation of array bound checker. In *Proceedings of the 4th ACM Symposium on Principles of Programming Languages*, pages 132–143.

[Tofte, 1990] Tofte, M. (1990). Type inference for polymorphic references. *Information and Computation*, 89:1–34.

[Xi and Pfenning, 1997] Xi, H. and Pfenning, F. (1997). Eliminating array bound checking through dependent types. Available electronically from `http://www.cs.cmu.edu/~fp/papers/bounds97.ps.gz`.

Appendix A

Proofs of confluence

In this appendix we prove confluence of SIZE and VEC. The two proofs are very similar in structure and we therefore give a detailed description of only one of them – the one for SIZE.

A.1 Confluence of SIZE

Confluence of SIZE directly follows the confluence of any regular combinatory reduction system, a theorem proved in [Klop, 1980]. We begin by giving the definition of a combinatory reduction system (CRS from now on) and stating the confluence theorem for regular CRS's. We then show that the rewrite relation $\rightarrow_{\mathsf{sz}}$ on SIZE terms can be embedded in a regular CRS. The confluence of $\rightarrow_{\mathsf{sz}}$ will then follow.

We start with a number of definitions, all of which can be found in [Klop, 1980].

Definition A.1.1 The *alphabet* of a CRS Σ consists of

1. a countably large set $Var = \{x, y, z, \ldots\}$ of variables.
2. a set $C = \{c_1, c_2, \ldots\}$ of constants.
3. a set $Mvar = \{Z_i^k,\ k, i \in N\}$ of meta-variables.

In the third clause of the previous definition, the natural number k is the *arity* of Z_i^k.

Definition A.1.2 The *terms* of a CRS Σ are given by the following grammar

$$a, b \ ::= \ x \mid c \mid [x].a \mid a\ b$$

where $x \in Var$ is a variable and $c \in C$ a constant. In the last clause a can not be of the form $[x].a'$ for some a'.

Remarks

1. We shall use letters from the beginning of the alphabet to represent terms of a CRS.
2. Note that meta-variables do not appear in the definition of a term.
3. The construct $[x].a$ binds the variable x in a. The definitions of contexts, occurrences and substitution are analogous to those given in Chapter 2 and will not be repeated here.

A CRS provides us with a general variable-binding mechanism $[x].a$ which can be used to capture various variable-binding constructs. Thus, for example, one could model the abstraction construction of lambda calculi by the (CRS) term $\lambda\ ([x].a)$ (where $\lambda \in C$ is a constant). Analogously, the recursion construct can be represented by the term $\mathsf{rec}\ [f].a$, where $\mathsf{rec} \in C$.

Definition A.1.3 The *meta-terms* of a CRS Σ are given by the following grammar:

$$h \ ::= \ x \mid c \mid [x].h \mid h\ h \mid Z_i^k(\underbrace{h, \ldots, h}_{k\text{-times}})$$

where $x \in Var$, $c \in C$ and $Z_i^k \in Mvar$. As above, the construction of a meta-term of the form $([x].h')\ h$ is not allowed.

Remarks

1. A meta-term h is *closed* if it has no free variables. Note that a closed meta-term can still contain meta-variables – we only talk about free and bound *variables*.

2. We shall use $\equiv$ to denote the syntactic equality of both terms and meta-terms.

3. The *head* of a meta-term is defined as follows

   (a) the head of a variable or a constant is the variable or constant itself, respectively.

   (b) the head of $[x].h$ is $[x]$.

   (c) the head of $h_1\ h_2$ is the head of h_1.

   (d) the head of $Z_i^k(h_1, \ldots, h_k)$ is Z_i^k.

Definition A.1.4 A *valuation* ρ assigns to each meta-variable Z_i^k a term a and a list of k pairwise distinct variables $x_1, \ldots, x_k$. We will write valuations in the following form:

$$\rho(Z_i^k) = a(x_1, \ldots, x_k)\ .$$

A valuation ρ induces a mapping, also denoted ρ, from meta-terms to terms as follows:

$$\begin{aligned}
\rho(x) &\equiv x \\
\rho(c) &\equiv c \\
\rho([x].h) &\equiv [x].\rho(h) \\
\rho(h_1\ h_2) &\equiv \rho(h_1)\ \rho(h_2) \\
\rho(Z_i^k(h_1, \ldots, h_k)) &\equiv a[\rho(h_1)/x_1] \ldots [\rho(h_k)/x_k] \\
&\qquad \text{where } \rho(Z_i^k) \equiv a(x_1, \ldots, x_k)\ .
\end{aligned}$$

Definition A.1.5 A *combinatory reduction rule* ϕ is a pair of closed meta-terms h_1 and h_2 such that

1. the head of h_1 is a constant,

2. all meta-variables occurring in h_2 occur already in h_1,

3. all meta-variables Z_i^k in h_1 occur in the form $Z_i^k(x_1, \ldots, x_k)$ with variables $x_1, \ldots, x_k$ being pairwise distinct.

We shall usually write such a rule as $\phi : h_1 \to h_2$.

A reduction rule $\phi : h_1 \rightarrow h_2$ induces a reduction relation $\rightarrow_\phi$ on the set of terms as follows:

$$C[\rho(h_1)] \rightarrow_\phi C[\rho(h_2)]$$

for every context C and valuation ρ.

Thus the β reduction of a lambda calculus

$$(\lambda x.t)\ t' \ \rightarrow\ t[t'/x]$$

can be represented by the reduction rule

$$(\lambda\ [x].Z_0^1(x))\ Z_0^0 \quad \rightarrow \quad Z_0^1(Z_0^0)\ .$$

Similarly, the reduction $\mathsf{rec}\ f.t \ \rightarrow\ t[\mathsf{rec}\ f.t/f]$ corresponds to

$$\mathsf{rec}\ [x].Z_0^1(x) \quad \rightarrow \quad Z_0^1(\mathsf{rec}\ [x].Z_0^1(x))\ .$$

Red(Σ) will denote the set of reduction rules of a CRS Σ. This set induces a reduction relation $\rightarrow_\Sigma$ on the set of terms of Σ in the obvious way:

$$C[\rho(h_1)] \rightarrow_\Sigma C[\rho(h_2)]$$

for every rule $\phi : h_1 \rightarrow h_2$, context C and valuation ρ.

Example

Consider the combinatory reduction system Ω given as follows:

1. The set of constants of Ω is $\{\lambda, \mathsf{rec}, \mathsf{ifs}\} \cup C'$ where C' is the set of SIZE combinators.

2. Red(Ω) is the set of reduction rules corresponding to the base reduc-

tions of SIZE as follows:

$$
\begin{aligned}
(\lambda\ [x].Z_0^1(x))\ Z_0^0 &\rightarrow Z_0^1(Z_0^0) \\
\mathsf{fst}\ ((\mathsf{pair}\ Z_0^0)\ Z_1^0) &\rightarrow Z_0^0 \\
\mathsf{snd}\ ((\mathsf{pair}\ Z_0^0)\ Z_1^0) &\rightarrow Z_1^0 \\
\mathsf{pred\ zero} &\rightarrow \mathsf{err} \\
\mathsf{pred}\ (\mathsf{succ}\ Z_0^0) &\rightarrow Z_0^0 \\
(\mathsf{eq\ zero})\ Z_0^0 &\rightarrow Z_0^0 \\
(\mathsf{eq}\ (\mathsf{succ}\ Z_0^0))\ \mathsf{zero} &\rightarrow \mathsf{succ}\ Z_0^0 \\
(\mathsf{eq}\ (\mathsf{succ}\ Z_0^0))\ (\mathsf{succ}\ Z_1^0) &\rightarrow (\mathsf{eq}\ Z_0^0)\ Z_1^0 \\
(\mathsf{eq\ un})\ \mathsf{un} &\rightarrow \mathsf{zero} \\
(\mathsf{eq}\ ((\mathsf{pair}\ Z_0^0)\ Z_1^0))\ ((\mathsf{pair}\ Z_2^0)\ Z_3^0) &\rightarrow \\
(\lambda\ [x].(\lambda\ [y].((\mathsf{ifs}\ x)\ y)\ x))\ ((\mathsf{eq}\ Z_0^0)\ Z_2^0))&\ ((\mathsf{eq}\ Z_1^0)\ Z_3^0) \\
\mathsf{rec}\ [x].Z_0^1(x) &\rightarrow Z_0^1(\mathsf{rec}\ [x].Z_0^1(x)) \\
((\mathsf{ifs\ zero})\ Z_0^0)\ Z_1^0 &\rightarrow Z_0^0 \\
((\mathsf{ifs}\ (\mathsf{succ}\ Z_2^0))\ Z_0^0)\ Z_1^0 &\rightarrow Z_1^0
\end{aligned}
$$

There is the obvious embedding ω of SIZE terms to Ω terms given inductively as follows:

$$
\begin{aligned}
\omega(x) &\equiv x \\
\omega(c) &\equiv c \quad \text{for a combinator } c \\
\omega(\lambda x.t) &\equiv \lambda\ ([x].\omega(t)) \\
\omega(t_1\ t_2) &\equiv \omega(t_1)\ \omega(t_2) \\
\omega(\mathsf{rec}\ f.t) &\equiv \mathsf{rec}\ ([f].\omega(t)) \\
\omega(\mathsf{ifs}\ t\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2) &\equiv ((\mathsf{ifs}\ \omega(t))\ \omega(t_1))\ \omega(t_2)
\end{aligned}
$$

Since the reduction rules of Ω exactly correspond to those in SIZE, it is easy to see that whenever $t_1 \rightarrow_{\mathsf{sz}} t_2$ then $\omega(t_1) \rightarrow_\Omega \omega(t_2)$ and that whenever $\omega(t_1) \rightarrow_\Omega a$ then there exists a SIZE term t_2 such that $a \equiv \omega(t_2)$.

Definition A.1.6 A reduction rule $\phi : h_1 \rightarrow h_2$ is *left-linear* if no meta-variable occurs twice in h_1.

Definition A.1.7 We say that a meta-term h_1 *does not interfere with* a meta-term h_2 if whenever $\rho(h_1)$ is the subterm of $\rho(h_2)$ at occurrence $s \neq e$ (where ρ is a valuation), then there is an occurrence s' of a meta-variable Z

in h_2 such that s' is a prefix of s. In other words, $\rho(h_1)$ is a subterm of (the valuation of) a meta-variable.

The set $\mathrm{Red}(\Sigma) = \{\phi_i : h_i \to h'_i, i \in I\}$ of a CRS Σ in *non-ambiguous* if

1. $h_i \not\equiv h_j$ whenever $i \neq j$ and
2. h_i does not interfere with h_j for every $i, j \in I$.

The non-ambiguity of a set of reduction rules is the CRS's equivalent of having no critical pairs.

Definition A.1.8 Let Σ be a CRS such that every $\phi \in \mathrm{Red}(\Sigma)$ is left-linear and $\mathrm{Red}(\Sigma)$ is non-ambiguous. Then Σ is *regular*.

It is easy to see that the CRS Ω defined above is regular.

Theorem A.1.9 *Let Σ be a regular CRS. Then the relation $\to_\Sigma$ is confluent.*

Proof. Can be found in [Klop, 1980]. ∎

The previous theorem, when considered together with the remarks made after the example above, gives us the confluence of $\to_{\mathsf{sz}}$:

Theorem A.1.10 $\to_{\mathsf{sz}}$ *is confluent.*

A.2 Confluence of Vec

The proof of confluence of $\to_{\mathsf{vc}}$ is analogous to the one given in the previous section. We extend the set of reduction rules of Ω by the rules corresponding to those added in Vec:

$$
\begin{array}{rcl}
\mathsf{hd}\ (\mathsf{sing}\ Z_0^0) & \to & Z_0^0 \\
\mathsf{hd}\ ((\mathsf{cons}\ Z_0^0)\ Z_1^0) & \to & Z_0^0 \\
\mathsf{tl}\ (\mathsf{sing}\ Z_0^0) & \to & \mathsf{err} \\
\mathsf{tl}\ ((\mathsf{cons}\ Z_0^0)\ Z_1^0) & \to & Z_1^0 \\
\mathsf{length}\ (\mathsf{sing}\ Z_0^0) & \to & \mathsf{succ}\ \mathsf{zero} \\
\mathsf{length}\ ((\mathsf{cons}\ Z_0^0)\ Z_1^0) & \to & \mathsf{succ}\ (\mathsf{length}\ Z_1^0)
\end{array}
$$

Theorem A.2.1 $\to_{\mathsf{vc}}$ *is confluent.*

Proof. It is easy to see that the resulting CRS is still regular, and we can again embed the reduction relation $\to_{\mathsf{vc}}$ to this system in the same way as above. $\to_{\mathsf{vc}}$ is thus confluent. ∎

Appendix B

Implementation

The methods and techniques presented in this thesis have been implemented in Standard ML [Harper et al., 1986]. The implementation is available from `http://linus.socs.uts.edu.au/~milan/research.html`. It comes in two files called `shape1.ml` and `shape2.ml` which contain the implementation of shape analysis as defined in Chapters 4 and 5, respectively. The two implementations are discussed in the following sections.

B.1 Shape analysis to SIZE

The implementation of the shape analysis from Chapter 4 is available from `http://linus.socs.uts.edu.au/~milan/shape1.ml`.

The implementation does not support types (since both SIZE and VEC are simply typed, adding types would only increase the complexities of term representations without significantly helping the programmer). The programmer is thus responsible for ensuring that ML expressions correspond to well-typed terms. Moreover, since we showed in Chapter 3 that SIZE is a sublanguage of VEC, both SIZE and VEC terms are represented by the same ML datatype `term`.

The main implemented operations are evaluation (`eval:term -> term`) and shape analysis (`shape:term -> term`). Also, a pretty printer has been implemented as `pretty:term -> string`. The operation `showev:term -> string` evaluates a term and then prints out the resulting value using the

pretty printer.

A suite of SIZE and VEC terms (including all the examples introduced in the text) is included in the code. Thus, for example, the matrix

$$\mathsf{mat1} \equiv \mathsf{make}\ \tilde{}2\ (\mathsf{make}\ \tilde{}3\ 1)$$

defined in Section 3.3 is represented by the expression

```
val mat1 = app2h(make,pos 2,app2h(make,pos 3,dint 1));
```

(a description of the term constructors and operations used above can be found in the source file). If we evaluate `mat1` by running

```
showev mat1;
```

the program returns the following (expected) result

```
[[1,1,1],[1,1,1]]
```

We can get the shape of mat1 by running

```
showev (shape mat1);
```

which returns

```
<~2,<~3,!>>
```

Additional, more technical information can be found in the source file.

B.2 Shape analysis to SIZE_C

The implementation of the shape analysis from Chapter 5 is available from `http://linus.socs.uts.edu.au/~milan/shape2.ml`. It is built along the same lines as the one described in the previous section. The main addition is the operation `simplify:term -> term` which reduces a SIZE_C term to a normal form under the $\rightarrow_{\mathsf{sc}}$ reduction. The operation `display:term -> string` then prints out the simplified shape of a VEC term. Thus, for example,

```
display multiply;
```

returns the simplified shape of matrix multiplication:

```
Lam a.Lam b.
        check fst (snd (a))=fst (b)
        then <fst (a),<fst (snd (b)),!>>
```

Again, an example suite and more detailed technical information is included with the code.