Alex - A Paradigm for Expressing and Compiling Matrix Functions

Eugene Kenneth Ressler
Ph.D Thesis

93-1361
June 1993

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

ALEX — A PARADIGM FOR EXPRESSING AND COMPILING MATRIX FUNCTIONS

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Eugene Kenneth Ressler

May 1993

ALEX — A PARADIGM FOR EXPRESSING AND COMPILING MATRIX FUNCTIONS

Eugene Kenneth Ressler, Ph.D.
Cornell University 1993

This work presents formal and practical tools to support the Alex paradigm for expressing and compiling matrix functions. Alex programs are recursive definitions over matrices with the same flavor as the elegant FP language of Backus. Many useful matrix algorithms can be expressed in both FP and Alex without any explicit index arithmetic. While FP is very difficult to compile to fast numerical codes, we show that Alex functions admit compile-time analyses and code generation techniques with efficient results. In particular, we give a type system that expresses matrix sizes and shapes, a sound and complete inference algorithm for these types, and a code generation algorithm that is space-efficient—storage is allocated only for user function parameters and results. Finally, we give an algorithm that determines when it is safe to replace call by value array parameters in the compiled code with call by reference. A compiler using these techniques generates C code from Alex programs which is at least as fast as code one would write by hand for some small problems. We conclude with discussion of how the fundamental techniques given here relate to previous work and how they should be extended to a general functional language or incorporated as a feature of an existing one.

Biographical Sketch

Gene Ressler was born in Allentown, Pennsylvania on July 20, 1956. He graduated from Northampton High School, Northampton, Pennsylvania in 1974, and accepted an appointment to the United States Military Academy at West Point, New York. On June 7, 1978, he graduated with a Bachelor of Science degree and was commissioned a Second Lieutenant of the Army Engineers. He served as a platoon leader, executive officer, intelligence officer, and commander of Company B of the 79th Engineer Battalion, a unit whose peacetime mission was military construction in Germany. He was married to Friederika Campbell on August 2, 1980 in Karlsruhe, and their first child Daniel was born on May 11, 1981 in Heidelberg. They left Germany in December, and at their next station, Fort Belvoir, Virginia, daughter Julia was born on July 23, 1982. Gene received a Masters Degree in Computer Science from the University of California at Berkeley in July 1985, and taught and did research in computer-assisted topographic mapping at West Point until July, 1988. For the following year, stationed at Seoul Korea, he managed the facilities maintenance budget and design staff for Eighth Army. From July, 1989 to the present date, he has worked toward the PhD in Computer Science at Cornell University, Ithaca, New York.

To Freddie, Daniel, and Julia,

and especially to Mom and Dad.

Acknowledgements

I am grateful to the Army for the incredible opportunity of my time at Cornell. I deeply appreciate the help and support of my advisors Jack Muckstadt, Keshav Pingalli, and, most especially, Dexter Kozen throughout this research. Finally, I thank my family, who cheerfully endured me as less of a husband, father, son, and brother than I really should have been during this time.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

This thesis formalizes and gives implementation techniques for the functional programming paradigm of Alex [KTC+87], a graphical programming environment for matrix functions. It also shows how the paradigm extends to functional languages in general. The Alex paradigm is based on a new denotational definition of matrix values, called s-vals, that supports a small but expressive collection of primitives for mapping functions over matrix elements. For illustration, we give a small, Alex-representative textual language, **svexp**, which uses these primitives for all computation. We show **svexp** sufficient to express many common numerical algorithms using no declarations or indexing expressions. This is achieved through a type system that includes a notion of the sizes and shapes of s-vals. An associated inference algorithm eliminates the need for run time representations of these types. We derive a code generator for **svexp** expressions, finding that for non-recursive definitions, the well-known functional *aggregate update problem* of superfluous memory allocation and copying is easily avoided. The recursive case is solved by using abstract interpretation techniques to determine when function results can be evaluated into the storage provided for arguments. Policy decisions that trade space for speed and available parallelism are explicit black boxes in the code generator that may be filled with resource-dependent heuristics. Finally, we show how the work on **svexp** extends to Alex and to other, more general, languages, including those with function values.

In a global sense, this thesis is a step in the ascendance of functional programming techniques from research topic to useful tool. Matrix calculations are certainly vital to scientific and business applications, where the advantages of functional techniques are relevant. Yet matrices have been addressed in only a limited way by functional language research and implementations, perhaps because only recently have the performance problems regarding simpler data types appeared surmountable. Our goal is to describe the key additional problems that matrices introduce, give a framework in which the problems can be solved, and give solutions for what we judge to be some of the more important ones, as we now explain.

1.1 The Functional Programming Style

A program in a *functional language* is a series of function definitions in the form of equations, followed by an expression applying these definitions. The value of the expression is the answer returned by the program. Function application (of user-defined functions as well as primitives like +) is the sole operation. Variable names denote *values*, e.g. numbers or sequences which, once associated with the variable in question, do not change. The process of value-variable association occurs only during function calls, when argument values are *bound to* formal function parameters. Control statements also return values:

$$\text{if } B \text{ then } E_1 \text{ else } E_2$$

returns the value of E_1 if B has value *true* and E_2 for *false*.

In this way functional languages are distinguished from *imperative* ones, wherein variables stand for cells or blocks of cells in a *global store*. Cells hold values, which are changed by *assignment* operations. An assignment such as $i = i + 1$, a dubious mathematical assertion, has an imperative meaning — "replace the value in cell i with itself plus 1." In this setting, control statements do not return values. Instead they determine *which* assignments are executed in order to influence changes in the state of the store. Some part of the store finally becomes the answer returned by the program.

The notion that functional variables denote values rather than cells in storage is so essential that it is given the special name *referential transparency.*

1.2 Some History

John Backus inspired much research in functional languages with his seminal Turing Award paper [Bac78]. In it, he asserts that the imperative global store is problematic in several ways, which we examine as they apply to this work:

- The global store is humanly incomprehensible for programs of any size. This leads to programming errors.
- The store is difficult to handle in mathematical proofs of program and compiler correctness.
- The store introduces arbitrary constraints on the structure of data flow in programs that, in the end, limit performance.

Appropriately, he demonstrates with a code for vector inner product:

$$c := 0; \text{ for } i := 0 \text{ to } n - 1 \text{ do } c := c + a[i] * b[i]$$

In correspondence with the the points above, we note:

- The programmer is responsible for ensuring that arrays a and b are big enough, that n matches the sizes of the vectors stored therein, and that neither c nor i hold some later-required value when the code executes.

- It is cumbersome to prove this program computes the dot product. Indeed, one starts by assuming a and b are n-vectors, and this must be remembered when proving surrounding code correct.

- The fact that the products could be computed by many processors working independently and the sum accumulated through a tree-like network is hidden in the sequential assignments to c, where each successive value depends on the previous one.

In addition, behavior is unspecified when a and b are *not* n-vectors. A compiler for the program may assure that unassigned storage is set to zero and that so-called index range errors are detected at run time, but these do not help get the right answer unless we are very lucky.

Functional languages address these points as follows:

- Functional programs are essentially declarations of the problems they solve; if the problem specification is correct, a correct program is not far away.

- Functional primitives are easily axiomatized equationally, and many correctness and equivalence proofs can be carried out in a straightforward fashion using equational logic.

- Since a functional language cannot express storage locations, it cannot constrain data flow by storage dependence. Moreover, good functional languages are rich in operators (and ways to define new ones) that work simultaneously on many data elements rather than on one at a time, the so-called *mapping primitives*.

To demonstrate these points, here is Backus's functional definition of inner product:[1]

$$\mathit{InnerProduct}\ x \equiv \underbrace{(\text{Reduce} +)}_{(a)}\underbrace{(\text{ApplyToAll} *)}_{(b)}\underbrace{\text{Transpose}}_{(c)}\, x \tag{1.1}$$

A proper argument to this function is a sequence of two equally long sequences of numbers. Transposition (c) changes the pair of lists into a list of pairs. The multiplication of all pairs (b) reduces each to a number. Reduce+ (a) sums the resulting vector. It is easy to write formal axioms for Transpose, Reduce, and ApplyToAll that make a proof that this program computes dot product a simple exercise. Further, we can make error handling precise by e.g. having Transpose return a special value *error* if its inputs have unequal length, and by making all primitives *strict*. That is, on inputs of *error*, they return *error*, so the behavior of all functions is completely defined. No preconditions to the application of *InnerProduct* are required in correctness proofs.

1.3 Problems

We conclude that functional techniques are extremely worthwhile. Why then, after fourteen years since Backus's incitement to study, have they not flourished, except in the halls of computer science research? The answer is three words — performance, performance, and performance. A naive compiler of functional programs generates code that typically runs one to three orders of magnitude slower than equivalent, naively compiled imperative code. Why?

Functional language implementations tend to be slow exactly *because* the functional programmer has no control over storage, as this places the entire burden of storage management with the compiler and runtime system; and these tasks turn out to be formidable. For instance, even our tiny inner product example uses sequences

[1] Actually, Backus's language FP is a purely combinatorial functional language with no variables at all. We have added x for consistency with our own syntax. This is largely a matter of taste.

of arbitrary length. To represent these in a computer generally requires extra space for link pointers and a heap and garbage collector like Lisp. All entail significant overhead. Moreover, the pure function compiler has a *harder* job than one for (non-pure) Lisp. When a Lisp programmer wishes to concatenate sequences A and B, she is free to *assign* B to the tail of A to avoid building a fresh list for the result. This destroys the old value of A, a fact the programmer must remember and consider in the rest of the program. The functional program, however, *must* have the value of A (without B appended) available if it is ever needed after the concatenation is formed. Thus A (at least) must be copied as the price of general referential transparency.

The function call stack[2] is another problem. In functional programs, recursive functions are used to express looping computations. E.g., to sum the integers 1 to n, one might write

$$\mathit{sum}\ n \equiv \text{if } n = 1 \text{ then } n \text{ else } n + \text{sum } (n-1) \tag{1.2}$$

Naively compiled code for this program makes n recursive calls to *sum*, requiring an $O(n)$ stack.

Many other performance problems arise when e.g. functions can be bound to variables and passed as arguments, when evaluation is lazy, and for other language features. These are mostly independent of the discussion at hand, however, and we refer the interested reader to [FH88] or [PJ87].

Indeed, referential transparency can be *extremely* expensive if arrays are offered as a primitive data type. This has been explored in [HB85] and [Blo89], with the introduction of three operators for making, accessing, and updating vectors.

Definition 1.1 (Hudak's operators) *For positive integer n and function f over integers,* mkv $n\ f$ *returns v such that $v[i] = f\ i$ for $0 \leq i < n$. We call v an n-vector.*

[2]Or equivalently, continuation memory when this supplants a stack. See for instance [AT89].

Furthermore, for n-vector v and integer i, $0 \leq i < n$, *the expression* updt $v\ i\ x$, *returns a new vector* v' *such that for* $0 \leq j < n$, $v'[j] = v[j]$, *except* $v'[i] = x$. □

Updt is conceptually like assignment to array elements in imperative programs. One can increment elements 1 to n of a vector with this function:

$$inc\ v\ n \equiv \text{if } n = 0 \text{ then } v \text{ else } inc\ (updt\ v\ n\ (v[n]+1))\ (n-1) \tag{1.3}$$

This example tempts us to compile updt into an assignment to the storage allocated for the current value of v. As with list concatenation, however, assignment can compromise referential transparency. If v is a vector with $v[0] = 1$, and the left operand of + is evaluated first, then

$$(\text{updt}\ v\ 0\ 0)[0] + v[0]$$

has value 1 if references to v are transparent and 0 if updt assigns to the storage of v. We conclude that the compiled code for updt $v \ldots$ may have to copy all of v into fresh storage. If a compiler naively generates such copying code for *all* updates, (1.3) becomes a monster requiring $O(n^2)$ run time and storage. In general, if a program makes m transparent updates to an n-vector, the total cost is $O(mn)$ rather than $O(m)$ for imperative array assignments. This nasty behavior is known as the *aggregate update* problem.

Another important source of overhead comes from checking arguments so that *error* can be returned when they are not of proper form. For example, sequences must be distinguishable from atoms if Transpose is to check that its argument is a sequence of sequences. We may have to augment each value with explicit type information for this purpose. Type information consumes space and type checks consume processor cycles at run time.

1.4 Progress

Thus it appears at first glance that the advantages of convenience and mathematical clarity of functional programming over comparable imperative idioms are offset by asymptotically worse performance. Remarkably, in an environment where imperative compiler writers are striving to shave a few per cent from run times with elaborate optimizations, functional language proponents continue to attack the formidable problems we have described and many others with enthusiasm, and with steadily increasing success. We next consider some successful techniques that are related to the examples given above and to the techniques developed in the remainder of this thesis.

First, though there seems no way to avoid garbage collection completely, the best algorithms are quite efficient, requiring execution of just a few instructions per allocated object on average [App87,Ung84]. Most recently, work such as [JM89] gives analysis techniques that tell a compiler when it is safe to "compile in" [sic] the reuse of argument storage for function results in the case of cons structures (general binary trees). This can eliminate garbage collection in the functions where the analysis succeeds.

Unconstrained stack growth of functions that emulate loops is often easily prevented by a syntactic analysis known as *tail recursion removal.* Any function with form

$$\begin{array}{l} \text{function } f(x) \\ \quad \vdots \\ \quad \text{return(} f(a) \text{)} \\ \quad \vdots \\ \text{end} \end{array}$$

can be transformed to machine code of the form

$$\begin{array}{l}\text{function } f'(x)\\ \textit{start:}\\ \quad\vdots\\ \quad x := a;\ \text{goto } \textit{start};\\ \quad\vdots\\ \text{end}\end{array}$$

which has equivalent meaning,[3] but no recursive call or stack growth. [FH88] shows how this is done in a stack-based code generator. Another approach is to compile with a *continuation-passing* intermediate form [AT89], which makes tail recursion almost trivial to find and eliminate. Unfortunately, Example (1.2) is not in tail recursive form! Here the program can be transformed using algebraic identities to the following:

$$\begin{aligned} sum_r\ n\ r &\equiv \text{if } n = 1 \text{ then } (r{+}1) \text{ else } sum_r\ (n{-}1)\ (r{+}n) \\ sum\ n &\equiv sum_r\ n\ 0 \end{aligned}$$

This version is tail recursive.

The aggregate update problem is tackled in the same work of Hudak and Bloss that introduced the vector operations mkv, $\cdot[\cdot]$, and updt. Their approach is to analyze the uses of updt in a program to determine when it is safe to compile them as assignments. Briefly, this is whenever the value of the vector argument to updt will never be needed after the update is performed. They note that imperative matrix programs translate in a line-for-line fashion to functional form as follows:

- mkv is used to simulate array declarations. Two and higher dimensional arrays are ostensibly represented as vectors of vectors.
- $\cdot[\cdot]$ is used in the same way as imperative array access.
- Tail-recursive functions simulate *for* loops.

[3]We mean equivalent in the extensional sense: For all x, $f'(x)$ returns r (or recurs forever) if and only if $f(x)$ returns r, (or loops forever).

Such programs submit completely to update analysis precisely because it is *impossible* to use the un-updated value of an imperative array after the array is updated by assignment.

Finally, [Mil78] and much subsequent work have given us practical *static type systems* and compile time *type inference* algorithms for functional languages. The latter compute with perfect certainty the types of definitions and subexpressions of a program, returning a type-annotated program or *fail* when no annotation exists. The only additional input is the types of primitives; e.g., $+$ has type $int \rightarrow int \rightarrow int$. Annotations may include *polymorphic types*, which at once represent many types with *type variables*. For instance, in

$$f\ x \equiv x$$

f will be annotated with $\forall\alpha.\alpha \rightarrow \alpha$, signifying that the type of the result is the same as the type of its input, whatever that may be. Such quantified types are *type schemes*. However,

$$g\ x \equiv x + 1$$

has type $int \rightarrow int$ owing to the application of $+$.

Type annotations often allow a compiler to avoid run time type checking completely, without compromising the safety of the compiled program. In the example above, any correctly typed application of g must be to an *int*, so the compilation of $x + 1$ may safely add without a check of the type of x. This saves both the run time of a type check and the space for type information.

1.5 Contribution of this Thesis

We are now in a position to consider a fresh approach to matrices in functional languages based on the qualities of existing approaches. Here are our observations:

- Backus showed the power of mapping arithmetic primitives over collections of numbers. However, his prototypical language can only represent matrices with sequences of arbitrary length. In implementations, these consume space for link pointers and require garbage collection. Moreover, determining the exact lengths of sequences in languages with general sequence-building primitives is an undecidable problem, and useful approximations are hard to compute. Thus, many checks for proper arguments, e.g. that Transpose is presented with a list of lists of equal length, must be performed at run time.

- Hudak proposed vector operators that presume underlying implementation as arrays in contiguous storage. They admit translation of an imperative program P to a functional form that compiles to efficient code; indeed P may be returned as output. We believe that though this is an important step forward as an analysis technique, it is a step backward for programming style simply because mkv, $\cdot[\cdot]$, and updt allow the programmer to bind a whole vector of values to a variable, then access and update these values randomly. This effectively simulates a piece of storage, realizing many drawbacks of the imperative global store in the functional model. As one instance, the correctness of (1.3) has the precondition that v is an m-vector with $m > n$. For another, when access to an undefined location of a vector occurs, we must resolve either to let it go, the equivalent of imperative programs ignoring references to cells on which no assignment has been performed, or ensure that *error* is returned. The former is clearly unacceptable, and the latter generally requires a run time check of every vector access. Finally, with individual element updates instead of function mapping as the fundamental operation available to the programmer, the parallelism inherent in many looping calculations is likely to be hidden in syntax.

This brings us to the contribution of this thesis, which is a way to use matrices themselves, rather than sequences or vectors that represent matrices, as a data type in functional languages. The approach has inherent advantages, which we intend to exploit:

- Matrices admit implementation as arrays, avoiding the pointer space and elementwise garbage collecting overhead of sequences.
- We can design flexible ways to map functions over matrices, making it easy for programmers to avoid elementwise access and update with their attendant difficulties. Moreover, mappings of common scalar and reduction primitives and nonrecursive user functions can be compiled so that copying is simply not required. With some work, this property extends to a useful class of recursive functions.
- The number and sizes of dimensions of a matrix can be represented in its type. A type inference procedure then annotates programs with the sizes of matrices produced by each subexpression, or returns *fail* when it can determine that dimension or size constraints inherent in the program could never be met at run time. Efficiencies in the compiled matrix code accrue as for other type-annotated programs; only a relatively few dimension and size constraints that are not resolved at compile time appear as checks in the compiled code.
- This style of program, where functions are mapped over matrices, submits to analyses that almost trivially disclose some optimizations which are difficult to perform on imperative programs.

1.6 A Preview of Structured Values

It is clear from the points of Section 1.5 that mappings of functions over matrix elements are essential to our paradigm. A simple way to provide such mappings would be with a matrix version of ApplyToAll in Example (1.1). The problem is that we would like to express many different kinds of mappings succinctly:

(a) Over the rows or columns of a rectangular matrix.

(b) Over the proper tri-diagonal band of a rectangular matrix.

(c) Over all contiguous 3×3 submatrices of a rectangular matrix.

(d) Over all $m \times n$ elements of a $p \times q \times m \times n$ four-dimensional matrix.

Operation (a) is a quintessential matrix algorithm building block, (b) is a typical sparse matrix operation, (c) arises in image processing problems. Finally, (d) is an unconventional formulation of a so-called *block matrix* operation, where a $(p \cdot m) \times (q \cdot n)$ two-dimensional matrix is considered as $p \times q$ blocks of size $m \times n$ (see [GVL83]). It is not hard to invent many other examples of useful mappings. Thus we might need many variants of ApplyToAll, each with its own definition.

Our approach is to consider the way we think about matrix calculations. One way to visualize vector inner product is as follows:

> Multiply all pairs formed from the the i-th element of a and the i-th element of b. Put the i-th product at position i of a vector and reduce this vector by addition to a scalar answer.

To amplify this idea, consider matrix multiplication:

> Form an (i, j)-th scalar result by applying inner product to the i-th row of A and the j-th column of B. Form the answer matrix such that the (i, j)-th scalar result is in row i and column j.

Conceptually, we are decomposing the input matrices into spaces of matrix components. In this case they are rows and columns, but other possibilities, even with more dimensions, are obvious. We call these *typical components.* Further, we say *spaces* rather than collections or sets because the order of components (in the matrix they came from) is significant, and is carried along in the decomposition. This allows a space of typical components to be recombined into a matrix later on, an operation called *projection.* Thus inner product uses two spaces of i-th scalars operated upon in i-th pairs by multiplication. Matrix multiply has a space of i-th and a space of j-th vectors where inner product is applied to all (i, j)-th pairs. In both, the application of a function (resp. multiplication and inner product) is simultaneous over many typical components. Inner product projects the space of multiplication results to form a vector. Matrix multiply projects the space of inner product results to form the result matrix.

Clearly, the components, being pieces of matrices, are matrices themselves. What may not be obvious is that the *spaces* are also best thought of as matrices. We refer to such a two level matrix — a matrix of matrices of values from some base set — as a *structured value* or *s-val.* In s-val calculations, the notion of a conventional matrix A (including a zero dimensional matrix, a conventional scalar) is replaced by an s-val containing exactly one component, the matrix A. Suppose A has m rows and n columns; then the s-val that represents A as a space of ith typical rows is an m-vector of typical components that are n-vectors. Moreover, the m-vector extends along a dimension we choose to label i. The n-vectors each extend along the dimension we choose to represent columns.

We will study five primitive operations on s-vals that, while not expressive enough to encode arbitrary random matrix access and update (as do Hudak's operators), are sufficient to express a large variety of mappings over matrices, and also computations that involve recursively splitting a matrix into smaller pieces. In very informal terms, the primitives are:

Typ Form typical components. Build an $n+1$-dimensional s-val of $m-1$-dimensional components from an argument with n dimensions of m-dimensional components.

Proj Project typical components. Project an n-dimensional s-val of m-dimensional components into an $n-1$-dimensional s-val of $m+1$-dimensional components. Proj is roughly the inverse of Typ.

Ext_0^0 (resp. Ext_0^1) Extract first (resp. last) components. Return the argument s-val with each component replaced by its own first (resp. last) component along some dimension. For example, extract the first row of each two-dimensional component of the argument.

Ext_1^0 (resp. Ext_1^1) Extract all but first (resp. all but last) components. Return the argument s-val, with each component replaced by itself, except its own first (resp. last) component is missing along a given dimension.

Join Adjoin pairs of components from two s-vals, V_1 and V_2. We describe this by example. Suppose V_1 has scalar components arrayed along its i and j axes and V_2 also has scalar components, but along its j and k axes. Then (Join Rows V_1 V_2) has three dimensions i, j, k; its (i,j,k)-th component is a vertical vector with two elements, the (i,j)-th element of V_1 and the (j,k)-th element of V_2.

It is convenient to define arithmetic in vector-reducing form, e.g. Red+ as well as the normal infix binary operators. Reduction is performed on all components of the single argument s-val simultaneously. Infix + behaves according to the following program fragment:

$$a + b \equiv \text{Red} + \text{ Rows (Join Rows } a\ b)$$

That is, we join the arguments into a 2-vector, then reduce by addition.[4]

[4]There is actually a slight difference between $a+b$ and Red+ Rows (Join Rows a b) in the final semantics of **svexp**. The former signals a type error unless both arguments have typical components are that are scalar. The latter produces a result for scalars *and* typical column vectors.

Extending the inner product example to s-vals, we have

$$\begin{array}{l} \mathit{InnerProduct}\ a\ b \equiv \\ \quad \text{Red+ Rows} \\ \qquad \text{Proj } i \text{ Rows} \\ \qquad\quad (\text{Typ Col } i\ a) * (\text{Typ Row } i\ b) \end{array}$$

Assume for the moment that the arguments a and b each contain only one typical component. Beginning with the last line, the i-th typical scalars of each input component are formed and pairs of these are multiplied to form i-th products. The projection of i onto Rows turns this space of scalars into a column vector, which is reduced by + to form the answer.

Of course, the inputs need not be so simple. Consider matrix multiply:

$$\begin{array}{l} \mathit{MatMult}\ A\ B \equiv \\ \quad \text{Proj } i \text{ Rows Proj } j \text{ Cols} \\ \qquad \mathit{InnerProduct}\ (\text{Typ Row } i\ A)\ (\text{Typ Col } j\ B) \end{array}$$

Again we form i-th typical rows of A and j-th columns of B. The semantics of *InnerProduct*, as hinted above, are such that the function acts simultaneously on all (i, j)-th pairs of vectors simultaneously.

1.7 A Larger Example

Many matrix algorithms use problem subdivision strategies that are conventionally hidden in explicit iteration. Consider LU factorization; one possible imperative coding in Pascal is given in Figure 1.1: Patient readers will glean that loop (a) with index j selects subproblems $A[j..n, j..n]$ of ever decreasing size. Loop (b) gets multipliers into the bottom $n - j$ elements of the j-th column. These form the L triangle of the factorization. Loops (c) and (d) do a rank one update of $A[j + 1..n, j + 1..n]$, which are the source of data for the next subproblem selected by loop (a). Careful inspection will show that while the iterations of loops (b), (c), and (d) are free to

```
procedure lu (var A: matrix; n: integer);
var i, j: integer;
begin
  for j := 1 to n-1 do begin                       (a)
    for i := j+1 to n do                           (b)
      A[i,j] := A[i,j]/A[j,j];
    for jj := j+1 to n do                          (c)
      for i := j+1 to n do                         (d)
        A[i,jj] := A[i,jj] - A[i,j] * A[j,jj]
  end
end
```

Figure 1.1: Pascal code for LU factorization

proceed simultaneously, each iteration of loop (a) depends on the values produced by the previous one, except the first, which depends on the input.

With s-vals, we express the dependency between the value of A in the j-th iteration and the $j+1$st with recursion. The other computations are typical element arithmetic. A code is given in Figure 1.2. Here Numof A *axis* n returns *true* if and only if A has size n along *axis*; it tests for the base case. If the test fails, then several extract operations cut A into four parts — the upper left element (*piv*), the rest of the top row and left column (*top_rt* and *bot_lft*), and the rest of A, with all but the first row and column (*bot_rt*). The two typical element computations are completed (*mults* and *updt*), then the recursive call factors *updt.* The final step is to join the multipliers and top row back onto the call result.

1.8 Alex

The syntax above is simple and verbose, as we do not wish to give formal semantics for a lot of syntactic sugar. Pattern matching (see e.g. [Rea89]), for instance, can

```
def lu A =
  if (Numof A Rows 1) or (Numof A Cols 1) then A
  else let
    top = Extr Fst Row A,        bot = Extr Allbut Fst Row A,
    piv = Extr Fst Col top,      top_rt = Extr Allbut Fst Col top,
    bot_lft = Extr Fst Col bot,  bot_rt = Extr Allbut Fst Col bot,
    mults = Proj i Rows (Typ Row i bot_lft / piv),
    updt = Proj i Rows Proj j Cols
             (Typ Row i Typ Col j bot_rt) -
                 (Typ Row i mults) * (Typ Col j top_rt)
  in Join Rows top Join Cols mults (lu updt)
```

Figure 1.2: **svexp** code for LU factorization

make s-val expressions more succinct and elegant, as it does in other functional languages. However, s-vals were actually conceived for a very sweet syntax indeed, the alexical syntax of the Alex programming environment [KTC+87]. In this system for matrix programming, one forms a function by manipulating a picture in a workstation window much as one might in a so-called paint program. Textual notes can be added for docmentation, but very little other typing is required. Values are denoted by boxes. S-val primitives correspond to manipulations of boxes. For instance, one forms 'Extr Fst Col V' and 'Extr AllBut Fst Col V' at once by clicking an Extract tool on the box that represents V. These are represented by a new, partitioned box that overlays V, but is offset a little down and to the right. Function calls are boxes labeled with the name of the called function, with small knobs to denote inputs and outputs. There is a library facility for user functions. When two different boxes represent the same value, they are given the same color. Conditionals are subwindows of the function window. Editing constraints on boxes in conditional windows prevent scope violations. One clicks on a button to see the true and false values of the conditional in alternation.

It turns out that our type inference algorithm can be adapted to work on incomplete Alex functions, returning a type if and only if a term may be extended to a complete Alex function body. The algorithm runs fast enough that it can be run after every user interaction as a function is created. Hence a user's request for an edit operation is honored only if it could result in a type-correct function. By induction, complete Alex function definitions are type-correct.

1.9 Generality

Clearly, though we have picked our primitives to exploit the kind of structure often found in matrix programs, not all programs will be expressible purely in this form. Some will need to compute indices and update matrix elements or components. For these instances, we fall back on s-val versions of Hudak's primitives, finding the extensions natural. For example, a function utilizing them behaves as expected when applied to a space of typical components.

A separate issue is expanding the set of primitives. Though our small set neatly expresses plain LU factorization, for instance, they cannot express partial pivots, which involve computing the row index of the maximum element of a column, and then swapping two rows at each iteration. Though we *could* use Hudak's primitives, explicit permutation primitives seem a preferable approach. We exclude such matters of taste from this thesis. Our intent is not to design a language, but to show how languages can be designed around s-vals with an expectation of good performance.

We also restrict most of our discussion to *first order* constructs; wherein functions cannot be passed as values. We also assume *call-by-value* argument passing semantics; all arguments are evaluated before control is passed to a called function. In principle, the s-vals presented here extend naturally to higher order languages and also to languages with lazy evaluation, but a treatment of the practical issues involved would easily make another thesis. Some directions to take are discussed in Chapter 7.

1.10 Denotational Semantics

Though s-vals and the primitives appear simple in the examples shown thus far, questions of how they should behave in general settings are somewhat involved and have deep interrelationships. As such, we choose a standard tool for unraveling programming languages — denotational semantics.[5] We find that a clean denotational definition provides natural answers to questions that are otherwise unclear. For instance, what should happen if we add a space of i-th pairs with one element drawn from each of two vectors as in the inner product example, but the input vectors have unequal lengths?

1.11 Outline

In Chapter 2, we introduce denotational semantics for matrices that support the formal definition of s-vals. We establish some basic consequences of this definition that will be of later use in designing a type system and inference algorithm. In Chapter 3, we give the denotational semantics of a small functional language, **svexp**, that uses s-vals for all of its calculations. We add theorems about primitives that are again helpful with types. Chapter 4 describes a type system and type inference algorithm that infers the numbers and sizes of dimensions of subexpressions in **svexp** programs, or reports *fail* when none exist. We are prove our types semantically sound; a program with a type yields an s-val result. We proceed to show the inference algorithm sound and complete (the latter in a restricted sense). Chapter 5 gives a code generator that allocates memory only for user function arguments and results. The code generation strategy is highly architecture-independent. It gener-

[5]The uninitiated reader should refer at least to [Rea89] on functional language semantics; [Sto77] is more comprehensive.

ates a single-assignment intermediate code that is portable to many architectures. Examples in the C programming language are given. Chapter 6 tackles the elimination of unecessary copying by the single assignment output of the **svexp** compiler. Finally, Chapter 7 covers related work and describes Alex in more detail. We draw final conclusions and give pointers toward work to be done.

Chapter 2

Matrices and Structured Values

In Section 1.6, we described an s-val as a *space* of *typical components*, giving a certain semi-formal, technical meaning to these terms through several examples. We then pointed out that a space of typical components is more precisely characterized as a matrix of matrices of base values. Indeed, we now replace the evocative terms space and typical component with more precise ones, *outer matrix* and *inner matrices.* The outer matrix is the s-val itself, an indexed collection of inner matrices that are ultimately acted upon "in parallel" by s-val arithmetic primitives. The inner matrices are closer to the standard notion of arrays — indexed collections of elements from some base domain. In **svexp**, the language described in Chapter 3, programs accept and return s-vals with exactly one inner matrix. Within a program, however, the Typ primitive splits single inner matrices into many for parallel computations, after which Proj assembles the results. It is curious that the definition of matrices that expresses this intuitive behavior, as described in Chapter 1, is itself rather nonintuitive. None the less, the definition and supporting notation and theorems allow us to define s-vals in the last section of this chapter. Finally, we give s-val semantics for classical matrix bracket notation. Not coincidentally, bracketed constants have values that are just (a subset of) the allowable **svexp** inputs and outputs mentioned above.

2.1 Domains

Matrices are built from *Scott domains* ([SS71,Sto77]) to ensure that recursive functions on s-vals have meaning.[1] We begin with representative flat base value domains from which s-val elements are taken:

$$
\begin{array}{lcll}
\mathbf{Bool} & = & \begin{array}{c} \mathit{true} \quad \mathit{false} \\ \diagdown \; \diagup \\ \perp \end{array} & \text{booleans} \\[2ex]
\mathbf{Int} & = & \begin{array}{c} \ldots \; \text{-1} \; 0 \; 1 \; \ldots \\ \diagdown\diagdown \mid \diagup\diagup \\ \perp \end{array} & \text{integers} \\[2ex]
\mathbf{B} & = & \mathbf{Bool} + \mathbf{Int} + \mathbf{B}_2 + \mathbf{B}_3 + \ldots & \text{other base types}
\end{array}
$$

For convenience, we refer to an arbitrary summand of $\mathbf{B}$ as $\mathbf{B}_i$, where $\mathbf{B}_0$ and $\mathbf{B}_1$ are synonymous with **Bool** and **Int** respectively.

We next select the fundamental representation for matrices. Conventionally, one picks a representation for vectors,[2] identifies zero-dimensional vectors with base values, then denotes an n-dimensional matrix recursively as a vector of $(n-1)$-dimensional vectors. Unfortunately, a recursive definition does not cleanly support our most fundamental operation — splitting a matrix into pieces along an arbitrary axis. We adopt an alternative choice for matrix representation, a function of tuples over the flat domain of natural numbers:

$$
\mathbf{N} \;=\; \begin{array}{c} 0 \; 1 \; 2 \; 3 \; \ldots \\ \diagdown\diagdown \mid \diagup\diagup \\ \perp \end{array} \qquad \text{flat natural numbers}
$$

Intuitively, n-tuples would be appropriate indices for an n-dimensional matrix. Instead, for reasons elaborated later, we choose *infinite* tuples of natural numbers. We define an infinite tuple as a function from positions, denoted by natural numbers, to values that are also natural numbers. Among index tuples, the ones that return $\perp$ for *any* input are all considered equally divergent. Hence, rather than the usual infinitely tall domain used for general functions, we arrange index tuples in a flat

[1] Scott originally used complete lattices, but we use complete partial orders (CPOs). The distinction is unimportant here.

[2] [Gor79] uses a list of index-value pairs.

CPO, **I**. All index tuples containing $\perp$ (that is, returning $\perp$ on any argument other than $\perp$ itself) are equal to $\perp$. All other tuples are greater than $\perp$ and otherwise incomparable.

Example 2.1 Even though index tuples have infinitely many elements, we will be interested in only finitely many of them. A bracketed list representation is sufficient to represent these, but we must append and elipsis to stand for the infinitely many values not denoted:

$$i = \langle 3, 5, 1, \ldots \rangle$$

Formally, i is a total function with $i\,0 = 3$, $i\,1 = 5$, and $i\,2 = 1$, and the rest of the map unspecified. □

Thus, if i is an index tuple, it is natural to speak of the value $(i\,p)$ as the *value of i at position p*. The notation above displays positions 0, 1, and 2. To avoid confusion with the *finite* tuples expressible in many languages, we henceforth call the elements of **I** *indices.*

We can now give the domains of that contain matrices as a proper subset:

$$\begin{array}{lcll} \mathbf{W}_r & = & \{\mathit{rangerr}\} & \text{range error value} \\ \mathbf{M}_{\mathbf{R}} & = & \mathbf{I} \rightarrow (\mathbf{R} + \mathbf{W}_r) & \text{contains matrices over domain } \mathbf{R} \\ \mathbf{M}_* & = & \Sigma_{\mathbf{R}}\, \mathbf{M}_{\mathbf{R}} & \text{contains all matrices} \end{array}$$

The special value *rangerr* is returned by a matrix when it is applied to an index that is out of range in a sense to be defined. Matrix values, **R**, may in principle be any domain, but we will be considering only certain cases elaborated in the rest of this chapter.

2.2 Matrices

Clearly, matrix domains as given above are overstocked with values that we would not consider matrices. A matrix should be a map on a finite collection of short index tuples. Yet here we have it defined as a map on an infinite set of infinite tuples. We need to restrict our semantics to a useful subset of M_{R}.

Definition 2.2 (Matrix) *$M \in \mathbf{M}_*$ is a* matrix *whenever it has properties 1, 2, and 3 below.* $\square$

Our first property says that indexing computations always succeed and that a matrix cannot magically make sense out of a divergent index expression.

Property 1 (Strong strictness) *For matrix M,*

$$i = \bot \textit{ if and only if } M\, i = \bot$$

$\square$

In particular, this leads easily to the following:

Theorem 2.3 *The matrices over any base domain plus $\bot$ form a flat (CPO) subdomain of* $\mathbf{M}_*$.

Proof. It is sufficient to show

$$M_1 \sqsubseteq M_2 \text{ if and only if } (M_1 = M_2 \text{ or } M_1 = \bot)$$

The *if* direction is trivially true, and so is the *only if* except for the case, call it $(*)$, $\bot \sqsubset M_1 \sqsubseteq M_2$. Here we need to establish

$$\forall\, i_1, i_2 \in \mathbf{I} \,.\, i_1 \sqsubseteq i_2 \quad \text{implies} \quad (M_2\, i_1) \sqsubseteq (M_1\, i_2)$$

For then $M_2 \sqsubseteq M_1$, and finally $M_1 = M_2$ as desired. By Strong Strictness, the implication is obvious when $i_1 = \bot$ or $i_2 = \bot$. Conversely, if $\bot \sqsubset i_1 \sqsubseteq i_2$, then $(M_2\, i_1), (M_1\, i_2) \sqsupseteq \bot$, also by Strong Strictness. Taken with $(M_2\, i_1) \sqsupseteq (M_1\, i_2)$, a consequence of $(*)$, we have $(M_2\, i_1) = (M_1\, i_2)$ because all these applications range over a flat base domain. $\square$

Next we associate with each matrix a finite *axis set* $A \subset \mathbf{N}$ and a *dimension function* $\alpha : A \to \mathbf{N}$. We write $M^{A,\alpha}$ to assert this association; it may be read as "M is a matrix with axis set A and dimension function α." Conceptually, A and α give the shape and size of the matrix. To formalize this, we define the following set.

Definition 2.4 *The set* $(\mathit{inrange}\, M^{A,\alpha})$ *is the set of indices* $i \in \mathbf{I}$ *satisfying:*

$$\forall\, a \in A \,.\, 0 \le i\, a < \alpha\, a$$

□

Example 2.5 Suppose $M^{A,\alpha}$, where $A = \{0\}$ and $\alpha\, 0 = 3$. Then $\mathit{inrange}\, M$ is all indices i that have values 0 to 2 at position 0, i.e., $0 \le (i\, 0) < 2$. For clarity, we can state this in an informal notation:

$$\mathit{inrange}\, M = \{\langle k, *, *, \ldots\rangle | 0 \le k < 3\}$$

where the angle brackets construct indices in the obvious way and $*$ denotes any natural number, and the elipsis extends them infinitely to the right. Similarly if $N^{A',\alpha'}$, $A' = \{1\}$ and $\alpha'\, 1 = 2$, then

$$\mathit{inrange}\, N = \{\langle *, k, *, \ldots\rangle | 0 \le k < 2\}$$

□

We can now insist that matrices be rectangular in a formal sense.

Property 2 (Rectangularity) *For matrix M and index i,*

$$i \in \mathit{inrange}\, M \text{ if and only if } M\, i \ne \mathit{rangerr}$$

□

Example 2.6 For this and all subsequent examples, we identify $0 \in \mathbf{N}$ with the notion of rows and 1 with columns. Then given M and N as above, M is a column 3-vector; it returns non-*rangerr* values if and only if it is presented with an index $\langle k, *, *, \ldots\rangle$, where $0 \le k < 3$. Likewise, N is a row 2-vector. □

Hence we say $M^{A,\alpha}$ *has size* n *on axis* a if $(\alpha\, a) = n$, and M *is* d*-dimensional* if $|A| = d$. The final property insists that if two indices agree at all elements of A, they must be mapped by $M^{A,\alpha}$ to the same value in $\mathbf{R}$.

Property 3 (Finiteness) *If* $M^{A,\alpha}$*, then for* $i_1, i_2 \in \mathbf{I}$*,*

$$(\forall\, a \in A \,.\, i_1\, a = i_2\, a) \quad \longrightarrow \quad M\, i_1 = M\, i_2$$

□

Example 2.7 Given M as above, M maps all indices $\langle k, *, *, \ldots\rangle$ to some $m_k \in \mathbf{R}$. It follows that the range of M has at most three elements (other than $\perp$ and *rangerr*).

□

Thus, a matrix $M^{A,\alpha}$ essentially ignores all the index positions not in A. Considering only the positions that *are* in A, M behaves just like a conventional matrix of mathematics.

Example 2.8 Suppose M and N as in the previous example, with values given by

$$M\langle k, *, *, \ldots\rangle = 2k + 1,\ 0 \le k < 3 \quad \text{and} \quad N\langle *, k, *, \ldots\rangle = 3k + 2,\ 0 \le k < 2$$

These correspond to the classical notation:

$$M = \begin{bmatrix} 1 \\ 3 \\ 5 \end{bmatrix} \quad \text{and} \quad N = \begin{bmatrix} 2 & 5 \end{bmatrix}$$

□

Intuitively, this example shows that our matrices have natural implementations as arrays in computer store, whereas most of the elements of $\mathbf{M}_*$ do not. Thus we will be careful to stick with language semantics that are *closed for matrices.* That is if an expression has a semantic value in some matrix domain (other than $\perp$), then it must satisfy Strong Strictness, Rectangularity, and Finiteness.

A question remains — why have we bothered using infinite structures for indices, just to define matrices that ignore all but a finite set of positions? The detailed answer is apparent only in the semantics of **svexp** given later, but the essence we can see right now: for any collection of matrices M_i, $1 \leq i \leq n$, the set $\bigcap_{i=1}^{n} \mathit{inrange}\, M_i$ is the *inrange* set of another matrix. This property lets us naturally specify the behavior of primitive and user functions of more than one argument when applied to s-vals with arbitrary outer matrices; i.e., the outer matrix of the result is non-*rangerr* for exactly the set of indices that is the *intersection* of the *inrange* sets of the arguments.

Example 2.9 Suppose $M_i^{A_i,\alpha_i}$, $i = (1,2)$, $A_1 = \{0\}$, $\alpha_1\, 0 = 3$, $A_2 = \{1\}$, $\alpha_2\, 1 = 2$. Then

$$\begin{aligned} \mathit{inrange}\, M_1 \cap \mathit{inrange}\, M_2 &= \{\langle k, \ldots\rangle \mid 0 \leq k < 3\} \cap \{\langle *, k, \ldots\rangle \mid 0 \leq k < 2\} \\ &= \{\langle k, l, \ldots\rangle \mid 0 \leq k < 3,\ 0 \leq l < 2\} \end{aligned}$$

Now if we construct $A = \{0, 1\}$, $\alpha\, 0 = 3$, $\alpha\, 1 = 2$, then any M such that $M^{A,\alpha}$ has $\mathit{inrange}\, M$ equal to the above set. □

This example generalizes:

Theorem 2.10 *1. Given n matrices* $M_i^{A_i,\alpha_i}$, $1 \leq i \leq n$, *let* $I = \bigcap_{i=1}^{n}(\mathit{inrange}\, M_i)$ *and* $A = \bigcup_{i=1}^{n} A_i$. *Then:*

$$I = \{i \mid \forall\, a \in A\,.\,(i\, a) < (\alpha\, a)\} \ \ \mathit{where}\ (\alpha\, a) = \min\, \{\alpha_i\, a \mid A_i \ni a,\ 1 \leq i \leq n\} \tag{2.1}$$

2. For any $M \in \mathbf{M}_*$ *such that* $M^{A,\alpha}$, *we have* $I = \mathit{inrange}\, M$.

Proof. By the definition of *inrange*,

$$\bigcap_{k=1}^{n} \mathit{inrange}\, M_k = \{i \mid \forall\, 1 \leq k \leq n, a \in A_k\,.\,(i\, a) < (\alpha_k\, a)\}$$

which is equal, by tedious set-builder manipulations, to (2.1). This proves part 1. Part 2 follows immediately from Rectangularity. □

2.3 λ and Other Tools

We now define some helpful notations and functions for dealing with matrices, indices, and s-vals.

2.3.1 λ as matrix constructor

Since we have used functions to represent matrices, a λ notation provides a natural way of describing them.

Example 2.11 Consider matrix M^A where $A = \emptyset$. By Property 3, M contains a single value, say 3. Then in mathematics M is conventionally described with

$$M = f \text{ where for all } i \in \mathbf{I},\ f\,i = 3$$

But it is far easier to write $M = \lambda i\,.\,3$. □

This use of λ is just a notational convenience for writing down semantic values in $\mathbf{M}_*$. Syntactic λ terms appearing in programs are entirely different entities.

Definition 2.12 *For any expression E, the following texts have equivalent meaning, provided the name f is not used elsewhere:*

1. *Function f, where for all $i \in \mathbf{I}$, $f\,i = E$.*
2. $\lambda i\,.\,E$

Furthermore, the meaning of 1 is taken as the usual one of function construction. □

Of course λ provides no guarantee that the constructed function is a matrix; e.g. $\lambda i\,.\,(i\,0)$.

2.3.2 Function patches

We will in many contexts need to describe a new function that is the same as an old one except at one value. For this we use the "patching" operator.

$$f\,[a/x] = f' \quad \text{where} \quad f'\,y = \begin{cases} f\,y, & y \neq x \\ a & y = x \end{cases}$$

An important example is patches to indices. If $i \in \mathbf{I}$, then $i\,[n/a]$ is the same as i except it has value n at position a.[3]

λ and index patches together let us patch matrices.

Example 2.13 Suppose $M^{\{0\},\alpha}$, i.e., M is a vertical vector. Then

$$M' = \lambda i\,.\,M\,i\,[(i\,1)/0] \tag{2.2}$$

is M transposed to form a horizontal vector. □

To see why this is true, consider any index $\langle *, k, \ldots \rangle$. Then

$$\begin{aligned} M'\langle *, k, \ldots \rangle &= (\lambda i\,.\,M\,i\,[(i\,1)/0])\,\langle *, k, \ldots \rangle \\ &= M\,(\langle *, k, \ldots \rangle\,[k/0]) \\ &= M\,\langle k, k, \ldots \rangle \\ &= M\,\langle k, *, \ldots \rangle \end{aligned}$$

The last step follows from $M^{\{0\},\alpha}$ and Finiteness. It is the convenience of expression exemplified here that makes the seemingly involved definition of matrices worth the trouble.

This example clarifies another feature of our matrices that first appeared in Example 2.8: they are strictly more expressive than the usual recursive definition of

[3] Of course, the function patch is itself a function. In fact, it stands for a family of functions, one for each domain of objects patched; indices, environments, etc. Conditionals are similar.

arrays (see Section 2.1). Vertical and horizontal n-vectors are distinct entities here, whereas there is no such distinction in a recursive definition. More generally, all index positions are first class citizens to our matrices. In recursive definitions, there is a total order on index positions implied by the left-to-right order in which they appear in programs. At best, subarrays of an n-dimensional array can be selected only by giving $k < n$ indices, which are assumed to index the first k dimensions in order:

Example 2.14 Here is a recursive array declaration in Pascal:

$$A : \mathrm{array}[0..9] \text{ of } \mathrm{array}[0..9] \text{ of Integer;}$$

The (p, q)-th element of A is expressed with $A[p][q]$, and the p-th row with $A[p]$, but there is no way to express the q-th column as a whole without explicitly copying it to a fresh 10-vector.

Conversely, given $M^{\{0,1\},\alpha}$, $(\alpha\, 0) = (\alpha\, 1) = 10$, which is equivalent to the definition to A above, the p-th row of M is $R = \lambda i\,.\, M\,(i\,[p/0])$ and the q-th column is just $C = \lambda i\,.\, M\,(i\,[q/1])$. The reader can easily verify $R^{\{1\},\alpha}$ and $C^{\{0\},\alpha}$. $\square$

This idea is generalized as follows.

Definition 2.15 *The* p-th component of matrix M on axis a *is* $\lambda i\,.\, M\,(i\,[p/a])$. $\square$

Of course this is finally the formal notion of typical components presented in Chapter 1.

2.3.3 Conditional values

To select one of two matrix values based on a truth value, we introduce the following:

Definition 2.16 *Operator* $\cdot \to \cdot, \cdot : \mathbf{Bool} \times \mathbf{M}_* \times \mathbf{M}_* \to \mathbf{M}_*$ *has the following value:*

$$B \to T, F = \begin{cases} T, & B = \mathit{true} \\ F, & B = \mathit{false} \\ \bot & B = \bot \end{cases}$$

□

We will freely use the same notation when T and F are drawn from some other domain. For instance, we make the following handy abbreviation:

$$\begin{array}{l} B_1 \wedge \ldots \wedge B_n \equiv B_1 \to \\ \qquad\qquad B_2 \to \\ \qquad\qquad\quad \ddots \\ \qquad\qquad\qquad B_n \to \mathit{true}, \\ \qquad\qquad\qquad \mathit{false}, \\ \qquad\qquad\quad \cdot^{\cdot^{\cdot}} \\ \qquad\qquad \mathit{false}, \\ \qquad \mathit{false} \end{array}$$

2.3.4 Promotion

Given M^A and an axis $a \notin A$, we will sometimes need to coerce M to a new matrix M' which is the same as M, except it has size one (rather than no size at all) on axis a:

Definition 2.17 *The* promotion *operator* $\uparrow : \mathbf{M}_* \times \mathbf{N} \to \mathbf{M}_*$ *is described as follows. If* $M^{A,\alpha}$*, then*

$$M{\uparrow}a = \begin{cases} M & \text{if } a \in A \\ \lambda i \,.\, (ia) = 0 \to (M\ i),\ \mathit{rangerr} & \text{otherwise} \end{cases}$$

□

Example 2.18 Suppose $M^{\{0\},\alpha}$ and $\alpha\, 0 = 3$, i.e. M is a vertical 3-vector. Then $M{\uparrow}0$ is M, $M{\uparrow}1$ is a matrix of three rows and one column, i.e. it has exactly one non-*rangerr* element along axis 1. Further, $M{\uparrow}1{\uparrow}2$ has three rows, one column, and also one non-*rangerr* element along axis 2, one *plane.* □

We can generalize this example:

Theorem 2.19 *If* $M^{A,\alpha}$, *then* $(M{\uparrow}a)^{A',\alpha'}$ *where* $A' = A \cup \{a\}$ *and*

$$\alpha' = \begin{cases} \alpha, & a \in A \\ \alpha\,[1/a], & a \notin A \end{cases}$$

Proof. Trivial if $a \in A$. If $a \notin A$, then we must verify (1) $M' \in \mathbf{M}_*$, (2) Strong Strictness, (3) Rectangularity with *inrange* M' given by A' and α', and (4) Finiteness. Indeed, (1) follows directly from the definitions of λ and M, (2) by case analysis and the definition of the conditional, and (4) from the finiteness of M. For (3), we observe from the given A' and α' that *inrange* M' is just all indices $i \in$ *inrange* M with value 0 at position a. For such i, $M\,i \neq$ *rangerr* by the rectangularity of M. Therefore $M'\,i \neq$ *rangerr* from the definition of promotion. This establishes the *if* part of Rectangularity for M'. Conversely, all other indices i' are either not in *inrange* M or have $(i'a) > 0$. Thus $M'\,i' =$ *rangerr*, establishing the *only if* part of Rectangularity. □

2.3.5 Matrix join

As a precursor of the definition of the **svexp** Join primitive, we now give auxiliary function $j_M\,c$ that, when applied to two matrices, say M_1 and M_2, concatenates them along axis c. Before a formal definition, let's see how it behaves in selected cases.

Example 2.20 Suppose M_1 and M_2 are both vertical 3-vectors as in the previous example, and $J = j_M\,0\,M_1\,M_2$. Then J behaves just like M_1 on indices of *inrange* M_1:

$$J\,\langle k, *, *, \ldots\rangle = M_1\,\langle k, *, *, \ldots\rangle,\ 0 \leq k < 3$$

Furthermore, the values of M_2 are "stacked above" those of M_1 on axis c:

$$J\,\langle k,*,*,\ldots\rangle = M_2\,\langle k-3,*,*,\ldots\rangle,\ 3 \le k < 6$$

and finally

$$J\,\langle k,*,*,\ldots\rangle = \mathit{rangerr},\ \ k \ge 6$$

We conclude J is a 6-vector. $\square$

Furthermore, $j_M\,c$ is overloaded in the sense that it works whether or not c is in the axis sets of its argument matrices:

Example 2.21 Suppose M_1 and M_2 of the previous example and $J = j_M\,1\,M_1\,M_2$. Then

$$\begin{aligned} J\,\langle k,0,*,\ldots\rangle &= M_1\,\langle k,*,*,\ldots\rangle && \text{if } 0 \le k < 3, \\ J\,\langle k,1,*,\ldots\rangle &= M_2\,\langle k,*,*,\ldots\rangle && \text{if } 0 \le k < 3, \text{ and} \\ J\,\langle k,l,*,\ldots\rangle &= \mathit{rangerr} && \text{otherwise} \end{aligned}$$

$\square$

This behavior is achieved by having j_M promote its arguments on axis c.

Definition 2.22 *Given matrices $M_1, M_2 \in \mathbf{M}_i$ for some i (i.e. both matrices are over the same base domain), let $M_1' = M_1{\uparrow}c$ and $M_2' = M_2{\uparrow}c$. By Theorem 2.19, these are matrices with known axis sets and dimension functions:*[4] ${M_1'}^{C'_1,\gamma'_1}$ *and* ${M_2'}^{C'_2,\gamma'_2}$. *Then*

$$j_M\,c\,M_1\,M_2 = \begin{cases} \lambda i\,.\,(i\,c) < (\gamma'_1\,c) \;\rightarrow\; M'_1\,i,\ M'_2\,i\,[(i\,c) - (\gamma'_1\,c)/c], & M_1, M_2 \ne \perp \\ \perp, & \mathit{otherwise} \end{cases}$$

$\square$

[4]We have used C and γ here rather than A and α here because the former will soon be adopted as the denotation for axis sets and dimension functions of *inner* matrices of s-vals, and it is to these that j_M is eventually applied.

The case for bottom-valued arguments preserves Strong Strictness, and Finiteness of the result given that the arguments have these properties. However, join results are usually *not* Rectangular. To get matrix closure, we will have to check the arguments for *conformability.* We forestall this until it is clear what to do with non-conformable arguments.[5]

2.4 S-vals

We can now formalize s-vals.

Definition 2.23 *A* structured value *or* s-val *is a matrix of matrices of elements from some flat base domain* $\mathbf{B}_k$. *Moreover,* simple *s-vals satisfy Property 4 given below.* □

Thus, the domains of s-vals (one for each base domain) must be

$$\mathbf{S}_k = \mathbf{M}_{\mathbf{M}_{\mathbf{B}_k}} = \mathbf{I} \to (\mathbf{W}_r + \mathbf{M}_{\mathbf{B}_k}) = \mathbf{I} \to (\mathbf{W}_r + \mathbf{I} \to (\mathbf{W}_r + \mathbf{B}_k)) \tag{2.3}$$

Recall that the $\mathbf{B}_i$ are our domains of base values, one per type.

We see an s-val is a curried function of two index arguments. Thus it is natural to speak of an s-val as an outer matrix of inner matrices. Respectively, two indices presented in order to an s-val are called *outer and inner indices.* We have expanded the domain equation above as a reminder that, for a s-val V, $V\,i\,j = \mathit{rangerr}$ holds if and only if $i \notin \mathit{inrange}\, V$ or else $j \notin \mathit{inrange}\,(V\,i)$. That is, we can get a range error if either the outer or inner index is not an element of *inrange* for the respective matrix.

The property required of simple s-vals in Definition 2.23 is just this: all inner matrices are of the same shape and size.

Property 4 *For any simple s-val* V*, there exists an axis set* C *and dimension function* γ *such that for all* $i \in \mathit{inrange}\, V$, $(V\,i)^{C,\gamma}$. □

[5] I.e. signal a type error.

We consider only simple s-vals. Triangular arrays and computations may be expressed in an extension of **svexp** allowing inner dimensions that vary systematically. This is a topic for future research. Itfollows from Property 4 that the size and shape of the outer and all inner matrices of an s-val V can be characterized by two axis set/dimension function pairs, one for the outer matrix, which we denote with A, α, and one for all inner matrices, C, γ. We assert that the outer matrix and inner matrices of $V \in \mathbf{S}_k$ satisfy Rectangularity with respect to A, α and C, γ by writing $V^{A,\alpha;C,\gamma}$. Portions of this unwieldy superscript are omitted whenever possible.

Example 2.24 An s-val with exactly one inner matrix of m rows and n columns $V^{A;C,\gamma}$ has $A = \emptyset$ (thus the outer dimension function is the empty map), $C = \{0, 1\}$ and γ such that $\gamma\, 0 = m$, $\gamma\, 1 = n$. $\square$

The use of λ to express matrices extends naturally to s-vals.

Example 2.25 Suppose $M = \lambda j\,.\,3$. That is, M is a matrix of empty axis set that maps all indices to 3. Further suppose V is an s-val of empty outer axis set, which maps all indices to M. Then $V = \lambda i\,.\,M = \lambda i\,.\,\lambda j\,.\,3 = \lambda ij\,.\,3$. $\square$

The last step above is a frequently used abbreviation for nested lambda terms.

Example 2.8 asserted the equivalence of certain s-vals having exactly one inner matrix and matrices modeled with classical bracket notation. Indeed our tools are now strong enough to give s-val semantics for matrix brackets:

Definition 2.26 *The* bracket constructor *for certain s-vals is given by:*

$$
\begin{aligned}
\begin{bmatrix} x \end{bmatrix} &\equiv \lambda ij\,.\,x \\
\begin{bmatrix} x_0 \\ \vdots \\ x_{m-1} \end{bmatrix} &\equiv \lambda ij\,.\,(j\,0) < m \;\rightarrow\; x_{(j\,0)},\ \mathit{rangerr} \\
\begin{bmatrix} x_0 & \cdots & x_{n-1} \end{bmatrix} &\equiv \lambda ij\,.\,(j\,1) < n \;\rightarrow\; x_{(j\,1)},\ \mathit{rangerr} \\
\begin{bmatrix} x_{0,0} & \cdots & x_{0,n-1} \\ \vdots & \ddots & \vdots \\ x_{m-1,0} & \cdots & x_{m-1,n-1} \end{bmatrix} &\equiv \lambda ij\,.\,(j\,0) < m \;\wedge\; (j\,1) < n \;\rightarrow\; x_{(j\,0),(j\,1)},\ \mathit{rangerr}
\end{aligned}
$$

□

It is not hard to prove that the constructed values are in fact s-vals with exactly one inner matrix.

We hasten to add that there are interesting uses for s-vals with more than two inner axes, and with inner axes other than 0 and 1. We have refrained from using them in examples thus far merely because they are harder to write down.

Chapter 3

Semantics of Svexp

This chapter gives the denotational semantics of a small language, **svexp**, that uses s-vals to express matrix programs with no index calculations, i.e., without Hudak's operators. We use **svexp** to show at once how the tools of denotational semantics apply to languages including s-vals and to explore the kinds of programs expressible using only the style of function mapping supported by s-vals.

An important step in developing **svexp** and any other language with s-vals is a proof that the semantics is closed for s-vals. That is, the semantic value of any expression is an s-val, a special value for "type error," or the so-called bottom value, which means that the expression loops forever. Such a proof ensures that values flowing through a program can be properly represented as blocks of contiguous storage, a vital consideration for efficient code generation, which is one of our main objectives. We prove the closure of **svexp** here as well as the foundational theorems for the type system developed in Chapter 4.

3.1 Abstract Syntax

We will work with the domain named **svexp** of terms with abstract syntax given in Figure 3.1. Some notes on the syntax: In general, n is used to represent arity.

$$\langle svexp \rangle \ ::= \ \text{def } \langle def \rangle^{+} \text{ in } \langle expr \rangle$$

$$\langle def \rangle \ ::= \ \langle fvar\text{-}n \rangle \ \langle bvar \rangle^{n} \ \equiv \ \langle expr \rangle$$

$$\langle expr \rangle \ ::= \ \langle const \rangle \mid \langle bvar \rangle \mid \langle fvar\text{-}n \rangle \ \langle expr \rangle^{n} \mid \langle prim\text{-}n \rangle \ \langle expr \rangle^{n}$$

$$\langle prim\text{-}1 \rangle \ ::= \ \text{Typ}\, c\, a \mid \text{Proj}\, a\, c \mid \text{Extr}_i^j\ c\ (i, j = 0, 1) \mid \text{Red} + c$$

$$\langle prim\text{-}2 \rangle \ ::= \ \text{Join}\, c \mid +$$

$$\langle prim\text{-}3 \rangle \ ::= \ \text{Cond}$$

Figure 3.1: Abstract syntax of **svexp**.

As a superscript over syntax, it means the syntax is repeated n times. The nonterminal $\langle const \rangle$ ranges over bracket terms as in the left hand sides in Definition 2.26. Binary + is given in prefix form for convenience. The nonterminals $\langle fvar\text{-}n \rangle$ and $\langle bvar \rangle$ represent the sets of n-ary *function variables* and *bound variables* respectively.[1] Such a separation is natural for first order languages and simplifies the semantics of function application:

Example 3.1 In the term *foo* $x\ y$, the grammar assures that *foo* is a function variable of arity two and that x and y are bound variables. □

An example of a program is given in Figure 3.2. For clarity in this work, we use numbers 7, 8, and 9 for outer axes to distinguish them from inner ones by inspection. The reader may find it helpful to consider these as synonyms for "i-th, j-th, and k-th" in the sense described in Section 1.6. For inner axes, 0 continues to stand for rows, 1 for columns, and 2 for planes. Thus Typ 0 7 x can be read "typical row i of

[1]This terminology conficts with the notion of bound variables in e.g. lambda calculus and logic, but is conventional in the literature for first order functional languages.

def
  scan-plus x $\equiv$ Proj 7 0
        + (Typ 0 7 x)(Typ 0 7 Extr_1^0 0 x)
  mat-add a b $\equiv$ Proj 8 1 Proj 7 0
        Red + 2
          Typ 0 7 Typ 1 8
            Join 2 a b
  max x y $\equiv$ Proj 7 0
        Cond (> (Typ 0 7 x) (Typ 0 7 y))
            (Typ 0 7 x)
            (Typ 0 7 y)
in
  Proj 7 1
    scan-plus Typ 1 7

$$\textit{mat-add} \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}$$

Figure 3.2: An **svexp** program.

x." In this program, *scan-plus* returns a vector one element shorter than its input, where the i-th element is the sum of the i-th and $(i+1)$-st elements of the input. Function *mat-add* adds rectangular matrices a and b by joining them like the bread slices in a sandwich along axis 2, taking a typical 2-vector plug from the sandwich (by two Typ operations), reducing this by addition, and projecting the result (twice) to again obtain a rectangular matrix. Finally, *max* scans its two vector inputs and puts the maximum of the i-th element of vector x and the i-th element of vector y into the i-th position of the result. The + in the abstract syntax is representative of all the usual scalar operations, including the test < in this code. The program body adds two matrices and applies *scan-plus* to each column of the result.

3.2 Semantic Groundwork

There are three kinds of expressible values in **svexp** semantics: s-vals, primitive functions, and user functions. Two special values are reserved for type errors. Roughly speaking, the first is for axis set and base type mismatches, and the second is for dimension mismatches. Hence the following domains include everything except user functions:

$$\mathbf{W}_t = \begin{array}{ccc} \mathit{typerr}_2 & & \mathit{typerr}_1 \\ & \diagdown \; \diagup & \\ & \perp & \end{array} \qquad \text{type error}$$

$$\mathbf{S} = \textstyle\sum_{k \geq 0} \mathbf{S}_k + \mathbf{W}_t \qquad \text{s-vals and errors}$$

$$\mathbf{P} = \mathbf{S} \rightarrow \mathbf{S} \qquad \text{primitive functions}$$

All domain sums coalesce bottom elements. Here $\mathbf{S}_k$ is as in Equation (2.3). It seems natural that $\mathbf{P}$ would also serve as the domain of user functions, but technical considerations lead us to a different approach. We will have user functions accept and return not s-vals, but simply matrices *over the base domains.* Allowing also for type errors in inputs and output, we have:

$$\mathbf{M} = \textstyle\sum_{k \geq 0} \mathbf{M}_{\mathbf{B}_k} + \mathbf{W}_t \qquad \text{matrices over base types and type errors}$$

$$\mathbf{F} = \mathbf{M} \rightarrow \mathbf{M} \qquad \text{user functions}$$

The input matrices come from the inner matrices of s-val arguments to which the function is applied. Similarly, the output matrices become inner matrices of the s-val result of the function application. It follows that the semantics of user function application must in concept "disassemble" argument s-vals, feed the pieces to the user function, and "reassemble" the answers into an s-val result.

Interpretation of variables in **svexp** programs is provided by *environments*, which are functions from variables to values. There is a set of these for bound variables and one for function variables.

$bve = bvar \rightarrow \mathbf{S}$	bound variable environments
$fve = fvar\text{-}n \rightarrow \mathbf{F}$	function variable environments

As a convention, β will range over bve and ϕ over fve. The binary function $+\!\!+$ catenates environments. That is, if $\beta = \beta_1 +\!\!+ \beta_2$ then $(\beta\, x) = (\beta_1\, x)$ if $x \in \mathrm{dom}\, \beta_1$ and $\beta_2\, x$ otherwise, and respectively for ϕ_1 and ϕ_2. Note that $+\!\!+$ is not commutative.

We compose the meanings of programs from the following semantic functions:

$\mathcal{I} : \mathbf{svexp} \rightarrow \mathbf{P}$	primitives
$\mathcal{E} : \mathbf{svexp} \rightarrow fve \rightarrow bve \rightarrow \mathbf{S}$	expressions
$\mathcal{D} : \mathbf{svexp} \rightarrow fve \rightarrow fve$	declarations
$\mathcal{P} : \mathbf{svexp} \rightarrow \mathbf{S}$	programs

It remains to elaborate their definitions.

3.3 Unchecked Semantics

It is appropriate to consider *unchecked semantics* for the **Svexp** primitives first, making assumptions as needed about the sizes and shapes of primitive inputs. We then complete the semantics by discharging these assumptions with so-called type checks that map assumption-violating inputs to a type error value.

3.3.1 Typ and Proj

The semantic function $\mathcal{I}$ interprets the s-val primitive operators as functions from s-vals to s-vals. It captures formally the behavior of the primitive operators described in Chapter 1. We begin with the Typ primitive. Recall (in the original, rough terms) that its purpose is to split the inner matrices of an s-val into components along some axis labeled, say, rows, and redistribute these along some outer matrix axis, labeled, say i. Hence we called the components "i-th typical rows" (see Section 1.6 and Definition 2.15). Using the foundation laid in Chapter 2, we can restate this notion precisely. Let c and a be natural numbers denoting an inner axis and an outer axis, respectively. Informally, the expression Typ $c\,a$ will be interpreted as the operator that takes an s-val V and produces a new s-val V' in which the inner matrices of V have been split along axis c and the components spread out along axis a. In other words, assuming that a is not already in the axis set of V, the k-th component of V' along axis a is an s-val of the same shape as V, but whose inner matrices are the k-th components of the corresponding inner matrices of V along axis c.

If a is already in the axis set of V, then the k-th component of V' along axis a is the same as the k-th component of V along axis a, except that the corresponding inner matrices of V along axis c are replaced by their k-th components in V'.

Using the definition of k-th component (Definition 2.15), if i is an index with $(i\,a) = k$, then we want $(V'\,i)$ to be the k-th component of $(V\,i)$ on c, which is

$$\lambda j\,.\,V\,i\,(j\,[k/c])$$

Thus we have

$$V'\,i\,j = V\,i\,(j\,[(i\,a)/c]).$$

Index patches, e.g. $(j\,[(i\,a)/c])$, are parenthesized here for clarity, but the parentheses are not written below; patches associate with the index to the left. This leads us to the formal definition

$$\mathcal{I}[\![\mathrm{Typ}\; c\, a]\!]V = \lambda ij\,.\,V\,i\,j\,[(i\;a)/c] \tag{3.1}$$

Example 3.2 Suppose V is an s-val of exactly one inner matrix $M^{C,\gamma}$, $C = \{0,1\}$, $\gamma\,0 = m$, $\gamma\,1 = n$; i.e. V represents a conventional $m \times n$ matrix. Then $\mathcal{I}[\![\mathrm{Typ}\,0\,7]\!]V$ is a new s-val that is an m-vector (along axis 7) of inner matrices, each an n-vector (along axis 1), a typical row of M. □

Proj effectively inverts this operation:

$$\mathcal{I}[\![\mathrm{Proj}\; c\, a]\!]\, V = \lambda ij\,.\,V\,i\,[(j\;c)/a]\,j \tag{3.2}$$

Example 3.3 Suppose V^A, where $7 \notin A$, then

$$\begin{aligned}
\mathcal{I}[\![\mathrm{Proj}\,7\,0]\!]\,(\mathcal{I}[\![\mathrm{Typ}\,0\,7]\!]\,V) &= \mathcal{I}[\![\mathrm{Proj}\,7\,0]\!](\lambda ij\,.\,V\,i\,j\,[(i\,7)/0]) \\
&= \lambda pq\,.\,(\lambda ij\,.\,V\,i\,j\,[(i\,7)/0])\,p\,[(q\,0)/7]\,q \\
&= \lambda pq\,.\,V\;\,p\,[(q\,0)/7]\,q\,[((p\,[(q\,0)/7])\,7)/0] \\
&= \lambda pq\,.\,V\;\,p\,[(q\,0)/7]\,q\,[(q\,0)/0] \\
&= \lambda pq\,.\,V\,p\,[(q\,0)/7]\,q \\
&= \lambda pq\,.\,V\,p\,q = V
\end{aligned}$$

□

The second to last step follows from $7 \notin A$ and Finiteness. This generalizes easily to the following.

Theorem 3.4 *Let V be an s-val with outer axis set A and $a \notin A$ be an outer axis number. Then* $\mathcal{I}[\![\mathrm{Proj}\; a\, c]\!]\,(\mathcal{I}[\![\mathrm{Typ}\; c\, a]\!]\,V) = V$. □

The astute reader will now ask, when is the condition $a \notin A$ not true, and what happens then?

Example 3.5 It is not hard to verify the following:

$$\begin{array}{l}\mathcal{I}[\![\mathrm{Proj}\,7\,0]\!]\\ \quad\mathcal{I}[\![\mathrm{Typ}\,1\,7]\!]\\ \qquad\mathcal{I}[\![\mathrm{Typ}\,0\,7]\!]\end{array}\begin{bmatrix}1&2&3\\4&5&6\\7&8&9\end{bmatrix}=\begin{bmatrix}1\\5\\9\end{bmatrix}$$

□

In general, composing the meanings of Typ c_i a for several inner axes c_i and finally a projection from a onto c yields a function that gets the "diagonal" along the axes c_i of each inner matrix of the operand.

3.3.2 Extr

Extract selects one or more adjacent components along a given axis c of each inner matrix of its argument. The simplest incarnation is $\mathrm{Extr}_0^0\, c$, which merely selects each zeroth component. The selected components become the inner matrices of the result. In the following equation, the λ term modifies its inner index argument so V always "sees" a zero in position c.

$$\mathcal{I}[\![\mathrm{Extr}_0^0\, c]\!]\, V = \lambda ij\,.\,V\, i\, j\,[0/c] \tag{3.3}$$

The complementary operator selects all but the first component. Here a λ term *increments* the inner index at position c:

$$\mathcal{I}[\![\mathrm{Extr}_1^0\, c]\!]\, V = \lambda ij\,.\,V\, i\, j\,[(j\ c)+1/c] \tag{3.4}$$

Thus each inner matrix of the result is the same as the corresponding inner matrix of the argument, except all components along axis c are "shifted down" by one position; the component at zero is lost. The superscript one versions similarly select the last and all but the last element, and we omit them.

3.3.3 Join

Join concatenates pairs of inner matrices from its arguments to form the inner matrices of its result as follows. Consider all $i \in \mathbf{I}$. If V_1 and V_2 are the Join arguments and V' the result, then $(V' i)$ is just $(V_1 i)$ concatenated with $(V_2 i)$, wherever both of these are non-*rangerr*. Concatenation is by matrix join given in Section 2.3.5.

$$\mathcal{I}[\![\text{Join } c]\!]\, V_1\, V_2 = \lambda i \,.\, i \in \mathit{inrange}\, V_1, V_2 \;\rightarrow\; j_M\, c\,(V_1\, i)\,(V_2\, i),\;\; \mathit{rangerr} \tag{3.5}$$

Of course we have still taken no precautions to ensure j_M is rectangular. This is another spot to mend with type checks.

3.3.4 Arithmetic

Reduction by the representative arithmetic operator + occurs simultaneously on all inner matrices, which are assumed to be vectors along axis c. To establish the upper limit for the reducing summation, part of the definition names the inner dimension function, which gives the length of the inner vectors:

$$\mathcal{I}[\![\text{Red} + \; c]\!]\, V = \lambda ij \,.\, \sum_{k=0}^{(\gamma\, c)-1} V\, i\, j\, [k/c] \text{ whenever } V^{\gamma} \tag{3.6}$$

Notice that j is used here merely because it is handy. Any index patched with k at c would work as well.

We pointed out in Chapter 1 that binary arithmetic can be expressed with Join and vector-reducing arithmetic. This suggests the following derivation, where the inner axis sets of V_1 and V_2 are assumed empty:

$$\begin{aligned}
\mathcal{I}[\![+]\!]\langle V_1, V_2\rangle &= \mathcal{I}[\![\text{Red} + \; 0]\!]\; \mathcal{I}[\![\text{Join}\, 0]\!]\, V_1\, V_2 && \text{(a)}\\
&= \lambda ij \,.\, \textstyle\sum_{k=0}^{(\gamma\, 0)-1} (\lambda i \,.\, i \in \mathit{inrange}\, V_1, V_2 \;\rightarrow \\
&\qquad\qquad j_M\; c\,(V_1\, i)\,(V_2\, i),\;\; \mathit{rangerr})\, i\, j\, [k/c] && \text{(b)}\\
&= \lambda ij \,.\, i \in \mathit{inrange}\, V_1, V_2 \;\rightarrow \\
&\qquad \textstyle\sum_{k=0}^{2-1} k < 1 \;\rightarrow\; (V_1\, i\, j\, [0/c]),\; (V_2\, i\, j\, [0/c]),\; \mathit{rangerr} && \text{(c)}\\
&= \lambda ij \,.\, i \in \mathit{inrange}\, V_1, V_2 \;\rightarrow\; (V_1\, i\, j) + (V_2\, i\, j),\;\; \mathit{rangerr} && \text{(d)}
\end{aligned} \tag{3.7}$$

This applies (3.5) and (3.6) above as well as Definition 2.22 and Theorem 2.19 in a straightforward manner. Axis 0 was used here for the join and reduction, but any other axis would serve as well, as is shown by the absence of zeros in (d). Note γ in (b) is the inner dimension of the j_M result. We presume a little by asserting $\gamma\, 0 = 2$ in (c), though it makes sense that joining two matrices with empty axis sets yields a 2-vector. We prove a generalization of this shortly.

3.3.5 Conditionals

Equations (3.7) ultimately offer an important insight. If we have an n-ary primitive *Op* that operates on base domains, the unchecked semantics for the corresponding "s-val version," Op, which acts on all inner matrices simultaneously, has a general form:

$$\begin{aligned} \mathcal{I}[\![\mathrm{Op}]\!]\, V_0 \ldots V_{n-1} &= \lambda ij \,.\, i \in \mathit{inrange}\, V_0, \ldots V_{n-1} \rightarrow \\ & \qquad Op\,(V_0\, i\, j), \ldots (V_{n-1}\, i\, j),\ \mathit{rangerr} \end{aligned} \tag{3.8}$$

We apply this reasoning to the base domain conditional operator to obtain:

$$\begin{aligned} \mathcal{I}[\![\mathrm{Cond}]\!]\, V_B\, V_T\, V_F &= \lambda ij \,.\, i \in \mathit{inrange}\, V_B, V_T, V_F \rightarrow \\ & \qquad ((V_B\, i j) \rightarrow (V_T\, i j),\ (V_F\, i j)),\ \mathit{rangerr} \end{aligned} \tag{3.9}$$

Here we assume the inner matrices of V_B map all indices to a single element of **Bool**, and that $V_T, V_F \in \mathbf{S}_k$ for some k.

Example 3.6 Recall Definition 2.26. Suppose we have

$$V_B = \begin{bmatrix} \mathit{true} \\ \mathit{false} \end{bmatrix}, \quad V_T = \begin{bmatrix} 1 \\ 2 \end{bmatrix}, \quad \text{and} \quad V_F = \begin{bmatrix} 3 \\ 4 \end{bmatrix}$$

Then form typical i-th components of all these and present them as conditional arguments, projecting the result:

$$\mathcal{I}[\![\mathrm{Proj}\,7\,0]\!]\,\mathcal{I}[\![\mathrm{Cond}]\!]\,(\mathcal{I}[\![\mathrm{Typ}\,0\,7]\!]\,V_B)\,(\mathcal{I}[\![\mathrm{Typ}\,0\,7]\!]\,V_T)\,(\mathcal{I}[\![\mathrm{Typ}\,0\,7]\!]\,V_F) = \begin{bmatrix} 1 \\ 4 \end{bmatrix}$$

This behavior is much like the vector merge instructions of high performance vector processors. On the other hand, let us now form i-th components of V_B, j-th components of V_T, and k-th components of V_F:

$$\begin{aligned} &\mathcal{I}[\![\mathrm{Proj}\,9\,0]\!]\,\mathcal{I}[\![\mathrm{Proj}\,8\,1]\!]\,\mathcal{I}[\![\mathrm{Proj}\,7\,2]\!]\,\mathcal{I}[\![\mathrm{Cond}]\!] \\ &\quad (\mathcal{I}[\![\mathrm{Typ}\,0\,7]\!]\,V_B)\ (\mathcal{I}[\![\mathrm{Typ}\,0\,8]\!]\,V_T)\ (\mathcal{I}[\![\mathrm{Typ}\,0\,9]\!]\,V_F) \end{aligned} \tag{3.10}$$

What is the value of this expression? Recall that, conceptually, Cond will return an (i,j,k)-th (i.e. $(7,8,9)$-th) typical component for each triple formed of an i-th, j-th, and k-th component of the inputs. The triples then consist of a truth value b from V_B, an integer t from V_T, and another integer f from V_F. The (i,j,k)-th answer component for the triple is just t if b is true, otherwise f. The i, j, and k axes are finally projected onto planes, columns, and rows of a three-dimensional result respectively. Thus the 0th plane corresponds to all the triples with $b = \mathit{true}$. Below we write the plane with these conceptual triples alongside the true answer, obtained by picking t in each triple.

$$\begin{bmatrix} (\mathit{true},1,3) & (\mathit{true},2,3) \\ (\mathit{true},1,4) & (\mathit{true},2,4) \end{bmatrix} \implies \begin{bmatrix} 1 & 2 \\ 1 & 2 \end{bmatrix}$$

Similarly for the *false* plane:

$$\begin{bmatrix} (\mathit{false},1,3) & (\mathit{false},2,3) \\ (\mathit{false},1,4) & (\mathit{false},2,4) \end{bmatrix} \implies \begin{bmatrix} 3 & 3 \\ 4 & 4 \end{bmatrix}$$

It is easy to verify this intuitive result formally. □

We have troubled with this long example because there was difficulty in reasoning *ad hoc* about how Cond should behave in such cases. Then Definition 2.2 gave us (3.8) and (3.9), and the difficulty went away.

The inner axis sets and dimensions of V_F and V_T match in this example. We must yet consider the case when they do not.

3.3.6 Expressions

The semantic function $\mathcal{E}$ gives the denotational meaning of expressions, composing the meanings of subexpressions in an appropriate way. For instance, we want (3.10) to be the meaning of any **svexp** expression

$$\begin{array}{l}\text{Proj}\,9\,0\;\text{Proj}\,8\,1\;\text{Proj}\,7\,2\;\text{Cond}\\ \quad(\text{Typ}\,0\,7\;E_B)\;(\text{Typ}\,0\,8\;E_T)\;(\text{Typ}\,0\,9\;E_F)\end{array}$$

where E_B, E_T, and E_F are terms with meanings V_B, V_T, and V_F respectively. Thus for an n-ary primitive, we apply the meaning of the primitive to the meanings of its arguments. Recall from Section 3.2 that ϕ and β are function and bound variable environments respectively. Then we have:

$$\mathcal{E}[\![\langle \textit{prim-n}\rangle\; E_1, \ldots E_n]\!]\,\phi\,\beta = \mathcal{I}[\![\langle \textit{prim-n}\rangle]\!]\;(\mathcal{E}[\![E_1]\!]\,\phi\,\beta)\ldots(\mathcal{E}[\![E_n]\!]\,\phi\,\beta) \tag{3.11}$$

Applying a user function is similar, but recall that user functions act on matrices over base domains rather than s-vals. The semantics of their application must encode application to each inner matrix explicitly. Again (3.8) is the appropriate idea. We need only "lift" it to the case where Op acts on matrices of base elements rather than base elements themselves:

$$\begin{aligned}\mathcal{E}[\![f\;E_1\ldots E_n]\!]\,\phi\,\beta \;&=\; \lambda i\,.\,i \in \textit{inrange}\,e_1,\ldots e_n \;\rightarrow\; f\,(e_1\,i)\ldots(e_n\,i),\;\textit{rangerr}\\ &\qquad \text{where } e_k = \mathcal{E}[\![E_k]\!]\,\phi\,\beta\\ &\qquad \text{and } f = \mathcal{E}[\![f]\!]\,\phi\,\beta\end{aligned} \tag{3.12}$$

The recursion in these rules is grounded at constants and variables. For a constant bracket term C, $\mathcal{E}[\![C]\!]\,\phi\,\beta$ is given in definition (2.26). Variable values come from the appropriate environment.

$$\mathcal{E}[\![x]\!]\,\phi\,\beta = (\beta\,x) \tag{3.13}$$

$$\mathcal{E}[\![f]\!]\,\phi\,\beta = (\phi\,f) \tag{3.14}$$

3.3.7 Declarations

The semantic function $\mathcal{D}$ assembles the meanings of function definitions to form the function variable environment. Each definition yields a singleton map for the defined function variable:

$$\begin{aligned} \mathcal{D}[\![f\ x_1 \ldots x_n \equiv E]\!]\,\phi &= [f/f], \quad \text{where} \\ f &= Y \lambda g y_1 \ldots y_n \,.\, M\, \mathcal{E}[\![E]\!]\ (\phi\,[g/f])\ (\bot\,[(S y_1)/x_1] \ldots\ [(S y_n)/x_n]) \end{aligned} \tag{3.15}$$

Here (Yf) takes f to its least fixed point, providing the so-called natural meaning of recursive terms. See [Sto77]. In fact the only uncommon feature of this equation is the interjection of auxiliary functions S and M. The former matches the input values x_k, which are matrices over base types, to the range of the bound variable environment, the s-val domain $\mathbf{S}$. Thus we want

$$S\,x = \lambda i\,.\,x \tag{3.16}$$

which changes a matrix over base types to an s-val of one inner matrix. On the other hand, if V is the s-val result of evaluating the function body, then $(M\,V)$, must be a matrix of base domain elements to provide the proper signature for the meaning of f. For this, we need to apply it to some index, but which one? Indeed, we achieve reasonable semantics for user functions only if the choice does not matter, i.e. if V is an s-val of empty axis set and has exactly one inner matrix. Thus user functions like

$$f\ x = \mathrm{Typ}\,0\,7\,x$$

whose bodies produce a space of components from inputs with only one component are not allowed. For the time being, we adopt this as an assumption. The type discipline of **svexp** will enforce it. This leads us to the following simple definition:

$$M\,V = V\,\mathbf{0} \tag{3.17}$$

where $\mathbf{0}$ is the (arbitrarily chosen) index that is zero at all positions.

$\mathcal{D}$ also accumulates successive function definitions:

$$\begin{aligned} \mathcal{D}[\![D_1, \ldots D_n]\!]\phi &= \phi' \mathbin{+\!\!+} \mathcal{D}[\![D_n]\!]\phi' \\ &\qquad \text{where } \phi' = \mathcal{D}[\![D_1 \ldots D_{n-1}]\!]\phi \end{aligned} \tag{3.18}$$

Finally, $\mathcal{P}$ gives the meaning of a complete program:

$$\mathcal{P}[\![\text{def } D_1, \ldots D_n \text{ in } E]\!] = \mathcal{E}[\![E]\!](\mathcal{D}[\![D_1, \ldots D_n]\!] \perp) \perp \tag{3.19}$$

The equation accumulates function definitions, then evaluates the body expression in the function variable environment they produce.

3.4 Full Semantics

We now set out to remove the simplifying assumptions made above that primitive and user function inputs are well-behaved, and that user function bodies produce values with only one inner matrix.

3.4.1 A primitive wrapper

The unary s-val primitives become well-defined on all inputs if we wrap the unchecked semantics in a filter that handles the cases we earlier assumed away. The filter functions like this:

- Propagate $\perp$, $typerr_1$, and $typerr_2$.
- Assert the argument is an s-val with certain axis sets and dimension functions.
- Test the axis sets and dimension functions for some property. If the test fails, return $typerr_1$. Otherwise, return the value given by the unchecked semantics.

We capture this behavior in an auxiliary function:[2]

Definition 3.7

$$\begin{aligned}\text{when } V^{A,\alpha;C,\gamma} \text{ has } P \text{ it's } V' \;=\; & (V = \perp) \rightarrow \perp, \\ & (V = typerr_1) \rightarrow typerr_1, \\ & (V = typerr_2) \rightarrow typerr_2, \\ & (V^{A,\alpha;C,\gamma} \wedge P) \rightarrow V',\; typerr_1\end{aligned}$$

□

[2]This form of auxiliary semantic function is borrowed from [Blo90]. We find it much more readable than others more common.

As a bonus, this maps all non-s-val inputs to $typerr_1$. However, we are out to prove that this never happens.

The interesting part of adding wrappers is in picking appropriate predicates P. Generally, the weakest acceptable P is dictated by our desire to obtain s-val closure for **svexp**. Recall that this involves showing that every program produces one of $\perp$, $typerr_1$, $typerr_2$, or an s-val. It follows that we must show all primitives closed for s-vals. The wrapper handles the details of propagating divergence and type errors. However, we still need to ensure that each primitive returns s-vals on s-val arguments. In doing this, we discover that the weakest P achieving closure can often be strengthened in a way that does not compromise expressiveness, but which prepares the ground for static type checking in the next chapter.

3.4.2 Typ and axis assertions for static typing

Wrapping the unchecked semantics for *Typ* with Definition 3.7 where $P \equiv c \in C$ yields

$$\begin{aligned} \mathcal{I}[\![\mathrm{Typ}\, c\, a]\!]V \;\; &\equiv \;\; \text{when } V^{A,\alpha;C,\gamma} \text{ has } c \in C \\ &\;\; \text{it's } \lambda ij\,.\, V\, i\, j\, [(i\; a)/c] \end{aligned} \tag{3.20}$$

The choice of P is important. Indeed, if $V^{A,\alpha;C,\gamma}$ and it is *not* true that $c \in C$ (there are no components to make typical), then $\lambda ij\,.\, V\, i\, j\, [(i\; a)/c] \;=\; V$. Thus we could have chosen $P \equiv \mathit{true}$ and (3.20) would still be closed. However, requiring $c \in C$ opens the door for static type inference — a term $(\mathrm{Typ}\; c\, a\, E)$ may be read as an assertion that the meaning of E has c in its inner axis set, whereas with $P \equiv \mathit{true}$, no such reading is possible.

This is comparable to the treatment of the cons primitive in dynamically typed Lisp *vs.* statically typed ML [MTH90]. In the former, lists may include any number of differently typed elements, so the code (cons a b) says nothing about the types of a or b. Conversely, ML lists are constrained to have elements of identical type, and the call (cons a b) adds a constraint to the overall type of the program: "b's type is 'list of T', where T is the type of a". Our constraint $c \in C$ sacrifices far less than the ML list constraint, however. There are many applications of dynamically typed lists, but no important ones for a new way to write an expression that is already written.

3.4.3 The other unary primitives

The wrapped version of Proj is:

$$\begin{aligned} \mathcal{I}[\![\text{Proj}\; c\; a]\!]V &= \text{when } V^{A,\alpha;C,\gamma} \text{ has } a \in A \wedge c \notin C \\ &\quad\ \text{it's } \lambda ij \,.\, V\, i\, [(j\; c)/a]\, j \end{aligned} \tag{3.21}$$

If $a \notin A$, the unchecked semantics again returns V: $\lambda ij \,.\, V\, i\, [(j\; c)/a]\, j = V$. Furthermore, if $a \in A$ and also $c \in C$, then the unchecked semantics gives an interpretation without apparent use—more than one outer axis is projected onto the same inner one. Hence an application of Proj asserts a type constraint: "V has outer axis a and no inner axis c," and again no useful expressive power is sacrificed.

Similarly for Extr we have:

$$\begin{aligned} \mathcal{I}[\![\text{Extr}_0\; c]\!]\, V &= \text{when } V^{A,\alpha;C,\gamma} \text{ has } c \in C \\ &\quad\ \text{it's } \lambda ij \,.\, V\, i\, j\, [0/c] \end{aligned} \tag{3.22}$$

$$\begin{aligned} \mathcal{I}[\![\text{Extr}_1\; c]\!]\, V &= \text{when } V^{A,\alpha;C,\gamma} \text{ has } c \in C \\ &\quad\ \text{it's } \lambda ij \,.\, V\, i\, j\, [(j\; c) + 1/c] \end{aligned} \tag{3.23}$$

Vector-reducing arithmetic requires checks that the operand's inner matrices are appropriate vectors of integers:

$$\begin{aligned} \mathcal{I}[\![\text{Red} +\; c]\!]\, V &= \text{when } V^{A,\alpha;C,\gamma} \text{ has } V \in \mathrm{S}_1 \wedge C = \{c\} \\ &\quad\ \text{it's } \lambda ij \,.\, \textstyle\sum_{k=0}^{(\gamma\, c)-1}\; V\, i\, j\, [k/c] \end{aligned} \tag{3.24}$$

Recall S_1 is the domain containing s-vals over integers.

3.4.4 S-val closure property for unary primitives

Let $a \dot{-} b = 0$ if $a < b$ and $a - b$ otherwise. We begin showing **svexp** is closed for s-vals by showing that the unary primitives return s-vals on s-val inputs:

Theorem 3.8 *Let U be a unary s-val primitive. If $V^{A,\alpha;C,\gamma}$ and $V' = \mathcal{I}[\![U]\!]$, then $V'^{A',\alpha';C',\gamma'}$, with A', α', C', and γ' as given in Table 3.1.*

Proof. The proof is by elementary consideration of primitive behavior on all possible s-val inputs. Consider $V = \text{Typ}\ c\ a\ V$. Thus

$$\begin{aligned} V' &= \text{when } V^{A,\alpha;C,\gamma} \text{ has } c \in C \\ &\quad \text{it's } \lambda ij\,.\,V\,i\,j\,[(i\,a)/c] \end{aligned}$$

V is an s-val. Hence it satisfies strong strictness and finiteness in both outer and inner matrices. Equation (3.20) preserves these properties by observation. Thus V' satisfies the properties as well.

It remains to show V' is rectangular in both inner and outer matrices with A', C', α', and γ' as given in the first line of Table 3.1. For this, we have

$$\begin{aligned} i \in \mathit{inrange}\,V' &\iff i \in \mathit{inrange}\,V \quad \text{and then} \quad j\,[(i\,a)/c] \in \mathit{inrange}\,(V\,i) \\ &\iff \forall b \in A\,.\,0 \leq (i\,b) < (\alpha\,b) \quad \text{and then} \quad 0 \leq (i\,a) < (\gamma\,c) \\ &\iff \forall b \in A \cup \{a\}\,.\,0 \leq (i\,b) < (\alpha'\,b) \\ &\qquad \text{where} \quad \alpha' = \alpha\,[\min\,\{(\alpha\,a), (\gamma\,c)\}/a] \end{aligned}$$

Thus $A' = A \cup \{a\}$ and $\alpha' = \alpha\,[\min\,\{(\alpha\,a), (\gamma\,c)\}/a]$. Similarly, for all $i \in \mathit{inrange}\,V'$ we have

$$\begin{aligned} j \in \mathit{inrange}\,(V'\,i) &\iff j\,[(i\,a)/c] \in \mathit{inrange}\,V \\ &\iff \forall b \in C - \{c\}\,.\,(j\,b) \in \mathit{inrange}\,V \end{aligned}$$

Thus $C' = C - \{c\}$ and γ suffices for γ' as desired. Proofs for the other primitives are similar. $\square$

Table 3.1: Axes and dimensions for unary primitives applied to $V^{A,\alpha;C,\gamma}$.

U *Primitive*	A' *Outer axes*	C' *Inner axes*	α' *Outer dimension*	γ' *Inner dimension*
Typ $c\ a$	$A \cup \{a\}$	$C - \{c\}$	$\alpha\,[\min\{(\alpha\ a),(\gamma\ c)\}/a]$	γ
Proj $a\ c$	$A - \{a\}$	$C \cup \{c\}$	α	$\gamma\,[(\alpha\ a)/c]$
$\mathrm{Extr}_0\ c$	A	$C - \{c\}$	α	γ
$\mathrm{Extr}_1\ c$	A	C	α	$\gamma\,[(\gamma\ c) \dot{-} 1/c]$
Red $+\ c$	A	$\emptyset$	α	$\emptyset$

3.4.5 Binary primitives

It is not hard to see that the unchecked semantics for Join (3.5) produces a matrix. However, the contents of the matix are not, in general, matrices. They may fail to satisfy Rectangularity. To rule out this case, we define a predicate on the inner axis sets and dimensions of the s-vals to be joined that is *true* if the result is rectangular. The definition is given in a compositional form: *conformable*$_1$ can be thought of as the "axis set part" of full conformability, which is tested by *conformable*$_2$.

Definition 3.9

$$(\mathit{conformable}_1\ c\ C_1\ C_2) = \begin{cases} \mathit{true} & \text{if } C_1 - \{c\} = C_2 - \{c\} \\ \mathit{false} & \text{otherwise} \end{cases}$$

$$(\mathit{conformable}_2\ c\ C_1\,\gamma_1\ C_2\,\gamma_2) = \begin{cases} \mathit{true} & \text{if } (\mathit{conformable}_1\ c\ C_1\ C_2) \text{ and} \\ & \quad \forall\, c \in C_1 - \{c\}\,.\,(\gamma_1\ c) = (\gamma_2\ c) \\ \mathit{false} & \text{otherwise} \end{cases}$$

A pair of matrices with axis sets and dimensions that satisfy conformable$_2$ *is called a conformable pair.* □

Then a conformable pair joins to form another matrix:

Theorem 3.10 *If* $M_1^{C_1,\gamma_1}$ *and* $M_2^{C_2,\gamma_2}$, *then* $(\mathit{conformable}_2\ c\ C_1\,\gamma_1\ C_2\,\gamma_2) = \mathit{true}$ *implies* $j_M\ c\,M_1\,M_2$ *is a matrix.*

Proof. It is easy to see that Definition 2.22 on page 34 for matrix join preserves strong strictness and finiteness. It remains to show the join is rectangular if and only if its arguments are conformable. Let $M = j_M\, c\, M_1\, M_2$ and also let M_1', M_2', γ_1' be as in Definition 2.22. Consider the set of indices

$$S = \{i \mid i \in \mathit{inrange}\, M_1' \quad \text{or} \quad i\,[(i\,c){-}(\gamma_1'\,c)/c] \in \mathit{inrange}\, M_2'\}$$

Straightforward case analysis yields

$$(M\,i) \neq \mathit{rangerr} \iff i \in S$$

Thus the proof finally devolves to showing that S is a valid *inrange* set when

$$(\mathit{conformable}_2\, c\, C_1\, \gamma_1\, C_2\, \gamma_2)$$

Indeed, when this is true, we have

$$S = \{i \mid \forall\, b \in C_1 \cup \{c\}\,.\, 0 \leq (i\,b) < (\gamma\,b)\}$$

where $\gamma = \gamma_1\,[(\gamma_1'\,c){+}(\gamma_2'\,c)/c]$. □

This proof also implies the following.

Corollary 3.11 *With M_1 and M_2 as in Theorem 3.10 and a conformable pair, let $M = j_M\, c\, M_1\, M_2$, where (by Theorem 3.10) $M^{C,\gamma}$. Then*

(a) $C = C_1\ \cup \{c\} = C_2 \cup \{c\}$

(b) $\forall\, c \in C - \{c\}\,.\,(\gamma\, c) = (\gamma_1\, c) = (\gamma_2\, c)$

(c) $(\gamma\, c) = \begin{cases} (\gamma_1\, c) + (\gamma_2\, c) \text{ if } c \in C_1, c \in C_2; & (\gamma_1\, c) + 1 \text{ if } c \in C_1, c \notin C_2 \\ (\gamma_2\, c) + 1 \quad\ \text{if } c \notin C_1, c \in C_2; & 2 \quad\quad\quad \text{if } c \notin C_1, c \notin C_2 \end{cases}$

□

With this, the full semantics of Join is:

$$
\begin{aligned}
&\mathcal{I}[\![\text{Join } c]\!]\ V_1\ V_2 \\
&\quad = \text{ when } V_1^{A_1,\alpha_1;C_1,\gamma_1} \text{ has } \mathit{true} \text{ it's} \\
&\qquad \text{when } V_2^{A_2,\alpha_2;C_2,\gamma_2} \text{ has } (\mathit{samebase}\ V_1\ V_2) \wedge (\mathit{conformable}_1\ C_1\ C_2) \\
&\qquad \text{it's } (\mathit{conformable}_2\ c\ C_1\ \gamma_1\ C_2\ \gamma_2) \rightarrow \\
&\qquad\qquad \lambda i\,.\, i \in \mathit{inrange}\, V_1, V_2 \ \rightarrow\ j_M\ c\,(V_1\, i)\,(V_2\, i),\ \mathit{rangerr}, \\
&\qquad\qquad \mathit{typerr}_2
\end{aligned}
\tag{3.25}
$$

Here *samebase* : $\mathbf{S} \rightarrow \mathbf{S} \rightarrow \mathbf{Bool}$ returns *true* if its two s-val arguments have the same underlying base type. This is clearly necessary for s-val closure, as the Join of, say, an s-val of integers and another of booleans would otherwise be some function not even in **S**. We then take pains to differentiate between type errors resulting from mismatched axis sets and those from mismatched dimensions. The distinction is helpful later for proving type inference sound.

Theorem 3.12 *The full semantics of the* Join *primitive is closed for s-vals. That is, if* $V_1^{A_1,\alpha_1;C_1,\gamma_1}$, $V_2^{A_2,\alpha_2;C_2,\gamma_2}$, *and* $V = \mathcal{I}[\![\text{Join } c]\!]\ V_1\ V_2$, *then* $V^{A,\alpha;C,\gamma}$, *with*

(a) $A = A_1 \cup A_2$,

(b) $\forall\, a \in A\,.\, \alpha\ a = \min\{(\alpha_1\ a),\ (\alpha_2\ a)\}$

(c) C *and* γ *as in Corollary 3.11.*

Proof. Strong strictness and finiteness are again preserved by the semantic equation (3.25) for both inner and outer matrices. It remains to verify rectangularity. The inner matrices are join results on conformable pairs; hence they are rectangular by Theorem 3.10 with size given by Corollary 3.11. These inner matrices, which are non-*rangerr* values, are returned when and only when the outer index is in *inrange* $V_1 \cap$ *inrange* V_2. This implies $V^{A,\alpha}$ as desired by Theorem 2.10. □

The full semantics for binary arithmetic follows by derivation from those of Join and vector-reducing arithmetic as in the unchecked case, (3.7) with the additional refinement of a check for empty inner axis sets for both operands.

$$\begin{aligned}\mathcal{I}[\![+]\!]\,V_1\,V_2 &= \text{when } V_1^{A_1,\alpha_1;C_1,\gamma_1} \text{ has } V_1 \in \mathbf{S}_1 \wedge C_1 = \emptyset \text{ it's}\\ &\quad\ \text{when } V_2^{A_2,\alpha_2;C_2,\gamma_2} \text{ has } V_2 \in \mathbf{S}_1 \wedge C_2 = \emptyset \text{ it's}\\ &\quad\ \lambda ij\,.\,i \in \mathit{inrange}\,V_1, V_2 \;\rightarrow\; (V_1\,i\,j){+}(V_2\,i\,j),\ \mathit{rangerr}\end{aligned} \tag{3.26}$$

Theorem 3.13 *If* $V_1^{A_1,\alpha_1;\emptyset,\gamma_1} \in \mathbf{S}_1$ *and* $V_2^{A_2,\alpha_2;\emptyset,\gamma_2} \in \mathbf{S}_1$, *then* $(\mathcal{I}[\![+]\!]\ V_1\ V_2)^{A,\alpha;\emptyset,\gamma}$ *where*

$$A = A_1 \cup A_2 \quad \mathit{and} \quad \forall\, a \in A\,.\,(\alpha\, a) = \min\{(A_1\, a), (A_2\, a)\}$$

Proof. Similar to Theorem 3.12. □

This is not surprising. When we add two spaces of typical scalars, we obtain another (appropriately shaped) space of typical scalars.

3.4.6 The conditional

The conditional requires care. We certainly have to ensure that the predicate is a boolean inner-scalar, but more is required to get s-val closure. Recall Example 3.6 and suppose

$$V_F = \begin{bmatrix} 3 & 5 \\ 4 & 6 \end{bmatrix}$$

instead of the value given. This presents a problem, as the final (unchecked) semantic value is, with an abuse of bracket notation:

$$\begin{bmatrix} 1 & \\ 4 & 6 \end{bmatrix}$$

More precisely, all inner indices $\langle 0, *, \ldots\rangle$ are mapped to a single value, 1, whereas the same cannot be said for $\langle 1, *, \ldots\rangle$, which map to one of 4, 6, and *rangerr*. Thus the unchecked semantic value violates inner Rectangularity and is not an s-val. One recourse is to insist that the inner axis sets and dimension functions of the true and false branches agree:

$$\begin{aligned}\mathcal{I}[\![\text{Cond}]\!]\, V_B\, V_T\, V_F \;\; = \;\; & \text{when } V_B^{A_B,\alpha_B;C_B,\gamma_B} \text{ has } V_B \in \mathbf{S}_0 \wedge C_B = \emptyset \text{ it's} \\ & \text{when } V_T^{A_T,\alpha_T;C_T,\gamma_T} \text{ has } \textit{true} \text{ it's} \\ & \text{when } V_F^{A_F,\alpha_F;C_F,\gamma_F} \text{ has } Q \text{ it's} \\ & \lambda ij\,.\, i \in \textit{inrange}\, V_B, V_T, V_F \;\rightarrow \\ & \qquad ((V_B\, i\, j) \;\rightarrow\; (V_T\, i\, j),\ (V_F\, i\, j)),\ \textit{rangerr}\end{aligned} \tag{3.27}$$

where

$$Q \equiv (\textit{samebase}\; V_T\; V_F) \wedge C_T = C_F \wedge \forall\, c \in C_T\,.\,(\gamma_T\, c) = (\gamma_F\, c) \tag{3.28}$$

The following is a nearly immediate consequence:

Theorem 3.14 *The full semantics of* Cond *given in Equations (3.27) and (3.28) is closed for s-vals. That is, if*

$$V_B^{A_B,\alpha_B;\emptyset,\gamma_B} \in \mathbf{S}_0, \quad V_T^{A_T,\alpha_T;C,\gamma} \in \mathbf{S}_k, \quad \textit{and} \quad V_T^{A_F,\alpha_F;C,\gamma} \in \mathbf{S}_k$$

for some k, then $(\mathcal{I}[\![\text{Cond}]\!]\; V_B\; V_T\; V_F)^{A,\alpha;C,\gamma} \in \mathbf{S}_k$, *where*

$$A = A_B \cup A_T \cup A_F \quad \textit{and} \quad \forall\, a \in A\,.\,(\alpha\, a) = \min\, \{(\alpha_B\, a),\ (\alpha_T\, a),\ (\alpha_F\, a)\}$$

Proof. Similar to Theorem 3.12. □

Let's look at where the search for closure has led us. First, Equations (3.27) and (3.28) have good properties for type inference and code generation. Type inference relies on computing representations of axis sets and dimension functions at compilation time. This cannot be done when the shape and size of conditional results depend on the value of the predicate. For code generation, if a compiler knows the exact size (by examining type information) of a conditional result, then it can allocate storage *a priori*, rather than, say, generating separate calls to a heap allocator for each branch.

These good features are offset by the definition's rather heavy-handed restriction of the **svexp** programmer. For instance, lists and vectors are both useful for representing sequences. Since s-vals generalize vectors, we would expect them to represent sequences as well. However, the conditional as given above allows us to write only programs where the true and false branches of conditionals produce (s-val representations of) sequences with lengths provably equal in any execution context.

To this there are two replies. First, we can adopt a narrow view. If some program condition determines the size of the output, then we might immediately question whether an s-val is the right data structure for the application. In other words, we can admit that s-vals are not a good way to represent general sequences.

The second reply is more flexible. There is an important generalization to Equation (3.27) that still admits s-val closure, though it makes **svexp** harder to compile. Observe that Rectangularity failed in the modified Example 3.6 because, intuitively, components from V_T and V_F were "shuffled" according to the inner truth values of the predicate to obtain the result. Since the components were not of the same size and shape, the shuffle result was not rectangular. Equation 3.27 fixed this by forcing all the components to be similar. An alternative course is to separate the case where no shuffle occurs, i.e. when the outer axis set of the predicate s-val is empty. Since there is only one (inner) truth value in this case, either V_T or V_F becomes the exact value of the conditional. It follows immediately that the result is an s-val, provided all the inputs are s-vals. We obtain such a semantics for cond by replacing (3.28) with

$$Q \equiv (\alpha_B = \emptyset) \tag{3.29}$$

In general, we may take Q as the disjunction of (3.28) and (3.29). This issue is somewhat of a Pandora's box that we examine again in Chapter 7. For the time being, we use the following semantic equation, which splits possible type errors into axis and dimension mismatches as we did for Join:

$$
\begin{aligned}
\mathcal{I}[\![\text{Cond}]\!]\, V_B\, V_T\, V_F \;=\;& \text{when } V_B^{A_B,\alpha_B;C_B,\gamma_B} \text{ has } V_B \in \mathbf{S}_0 \wedge C_B = \emptyset \text{ it's} \\
& \text{when } V_T^{A_T,\alpha_T;C_T,\gamma_T} \text{ has } \mathit{true} \text{ it's} \\
& \text{when } V_F^{A_F,\alpha_F;C_F,\gamma_F} \text{ has } Q_1 \text{ it's} \\
& Q_2 \;\rightarrow\; \lambda ij\,.\, i \in \mathit{inrange}\, V_B, V_T, V_F \;\rightarrow \\
& \qquad\quad ((V_B\, i\, j) \;\rightarrow\; (V_T\, i\, j),\, (V_F\, i\, j)),\; \mathit{rangerr}, \\
& \mathit{typerr}_2
\end{aligned} \tag{3.30}
$$

where $Q_1 \equiv (\mathit{samebase}\; V_1\; V_2) \wedge C_T = C_F$ and $Q_2 = \forall\, c \in C_T\,.\,(\gamma_T\, c) = (\gamma_F\, c)$. Since this merely specializes (3.27), we have

Corollary 3.15 *The full semantics of* Cond *given in Equations (3.30) is closed for s-vals. That is, Theorem 3.14 applies to these semantics also.*

3.4.7 Function application and construction

We give **svexp** strict function call semantics by propagating bottom for any input to the result. We also check the environment value for f to propagate "undefined function variable" errors.

$$
\begin{aligned}
\mathcal{E}[\![f\; E_1 \cdots E_n]\!]\, \phi\, \beta \;=\;& \text{when } e_1 \cdots e_n \text{ has } \mathit{true} \text{ it's} \\
& f = \bot \;\rightarrow \bot, \\
& \lambda i\,.\, i \in \mathit{inrange}\, e_1, \ldots, e_n \;\rightarrow\; f\,(e_1\, i) \cdots (e_n\, i),\; \mathit{rangerr} \\
& \text{where } e_k = \mathcal{E}[\![E_k]\!]\, \phi\, \beta,\; 1 \le k \le n \\
& \text{and } f = (\phi\, f)
\end{aligned} \tag{3.31}
$$

From this rule, we obtain the following closure theorem.

Theorem 3.16 *Suppose for given C_k, γ_k, and base domains $\mathbf{B}_{b_k}$, $1 \le k \le n$ there exist C, γ, and $\mathbf{B}_b$, such that $M_k^{C_k,\gamma_k} \in \mathbf{M}_{\mathbf{B}_{b_k}}$ implies $((\phi\, f)\, M_1 \cdots M_n)^{C,\gamma} \in \mathbf{M}_{\mathbf{B}_b}$. Let $V_k^{A_k,\alpha_k;C_k,\gamma_k} \in \mathbf{S}_{b_k}$ and $V = \mathcal{E}[\![f\; E_1 \cdots E_n]\!]\, \phi\, \beta$. Then $V^{A,\alpha;C,\gamma} \in \mathbf{S}_b$ where*

$$A = A_1 \cup \cdots \cup A_n \quad \textit{and} \quad \forall\, a \in A\,.\,(\alpha\, a) = \min\, \{(\alpha_k\, a) \mid 1 \le k \le n\}$$

Proof. Similar to Theorem 3.12. The requirements for dimensions of inputs of f replace the constraints of $conformable_2$. □

For function construction, Equation (3.15) needs only the addition of a type check in the auxiliary function M. Recall that M coerces the s-val of a function body to a simple matrix by applying it to the (arbitrarily chosen) zero index. We remarked in Section 3.3.7 that this is reasonable, i.e. that outer components are not lost, only if any other index would serve as well. In turn, this occurs only if the outer axis set of the body is empty. From this we have:

$$M\,V = \text{when } V^A \text{ has } A = \emptyset \text{ it's } (V\,\mathbf{0}) \tag{3.32}$$

3.4.8 S-val closure for svexp

The remaining unchecked equations for primitive application, variable reference, accumulation of definitions, and program values need no changes in the full semantics. The first big goal of our study is at hand:

Theorem 3.17 *The full semantics of* **svexp** *is closed for s-vals. That is, the value of an* **svexp** *program is one of* $\perp$ *(denoting* divergence, *i.e. an undefined variable error or infinite looping),* $typerr_1$ *or* $typerr_2$*, (denoting type errors), or else an s-val.*

Proof. As we have shown the primitives closed by Theorems 3.8, 3.12, 3.13 and 3.14, closure of **svexp** follows by elementary structural induction to prove the following assertions:

(a) Environments bind variables only to one of $\perp$, $typerr_1$, $typerr_2$, or an s-val.

(b) Function variables are bound in environments only to functions that are closed for matrices.

(c) **svexp** terms have values that are one of $\perp$, $typerr_1$, $typerr_2$, or an s-val.

Since a program is a term, this is sufficient. The only interesting case arises for user function definitions, where we must prove (b). Recall

$$\begin{aligned}&\mathcal{D}[\![f\ x_1 \ldots x_n]\!]\,\phi = [f/f], \quad \text{where} \\ &\quad f = Y\lambda g y_1 \ldots y_n\,.\,M\,\mathcal{E}[\![E]\!]\,(\phi\,[g/f])\,(\bot\,[(Sy_1)/x_1] \ldots\ [(Sy_n)/x_n])\end{aligned}$$

From the structure of our domains, we know

$$f = \bigsqcup_{i=0}^{\infty} f_i$$

where $f_0 = \lambda x_1 \ldots x_n.\bot$ and

$$f_i = \lambda y_1 \ldots y_n\,.\,M\,\mathcal{E}[\![E]\!]\,(\phi\,[f_{i-1}/f])\,(\bot\,[(S\,y_1)/x_1] \ldots\ [(S\,y_n/x_n])$$

Moreover, Theorem 2.3 assures us that if all the f_i are closed, then so is f. Certainly f_0 is closed. Then assuming it for f_{i-1}, we have (c) for the λ body by the overall structural inductive hypothesis (E is a smaller term than the definition wherein it lies), and thus (b) holds for f_i. □

Chapter 4

Types and Type Inference

The object of the foregoing chapters has been to define s-vals and **svexp**, and to show in a fundamental way that the definitions make sense. The final step was Theorem 3.17, which showed **svexp** programs do one of three things: diverge, generate a type error, or return a structured value. This is useful because s-vals, although they inhabit a general function domain, are neatly characterized by three finite features:

- The base element domain (name).
- Outer and inner axis sets.
- Outer and inner dimension functions.

We refer to these features collectively as the *extent* of an s-val. While proving s-val closure, we have given theorems for the syntactic constructs of **svexp** that allow us to precisely deduce the extent of the construct's semantic value from the extents of the values of its parts. An extent thus derived is a *valid extent.*

4.1 Overview

In this chapter, we develop an algorithm that deduces valid extents automatically. It manipulates expressions in a *type language*, which is a symbolic representation of extents. We want the algorithm to assign each subterm of a **svexp** program with a type whose meaning is a valid extent (or perhaps a set of them). This is akin to what is interchangeably called *type inference* and *type reconstruction* for conventional languages. However, s-vals introduce subtleties that do not otherwise occur.

4.1.1 History

In the pursuit of an extent inference algorithm, we follow a long line of distinguished research. Even early FORTRAN compilers deduced that the result of $1.2 + 3$ is a floating point number rather than an integer. There are good pragmatic reasons for this:

- *Convenience.* The compiler takes care of required details. Here the programmer is spared writing types for all subexpressions, which are needed for code generation.

- *Abstraction of hardware.* Types put a gloss on the details of physical implementation. Here the hardware has different internal models of numbers with and without fractional parts, which the type system casts in familiar terms of "real" and "integer" respectively.

Much subsequent work on type disciplines concentrates on convenience. At the extreme, the goal of modern *type theory* is to encode entire program specifications in types that are logical formulas provided by the programmer. A well-typed program is then certain to be correct.

On the other hand, our work is inspired by [Mil78] (and later work of many others), which shows how the types of all subterms in ML programs can be inferred by the compiler with no explicit programmer guidance. The compiler succeeds with type inference only if it is impossible for the program to suffer a type error later when the compiled program is run. Thus, ML compilers always have enough information to generate code without run time checks for the types of operands. This is somewhat surprising considering the power of ML. Functions can be *polymorphic*, e.g. a single sort function can sort lists of any type. Functions are also first class objects—they can be passed about as values much like integers. This contrasts with imperative languages like Pascal where monomorphic type expressions are a mandatory part of the syntax, and with dynamically typed functional languages like Lisp where no declarations are required, but type errors are discovered at run time.[1]

With this background, we can put our goals for extent inference in perspective. We seek the extension of ML's convenience benefits to matrix calculations. This includes removing, as much as possible, the programmer's burden of writing tedious and error-prone index arithmetic, loop bounds, and array declarations and offering her a matrix version of polymorphism. We also want to turn the full power of type inference back toward hardware abstraction, putting a gloss on the linear address space of memory in most modern processors.

4.1.2 The approach

The extent inference algorithm makes two passes over the program. The first discovers all axis set assignments and s-val element base domains consistent with the semantics, or stops and signals failure when no assignment exists. A language of *axis types* represents axis sets and element domains. *Typing rules* describe the axis types of **svexp** constructs in terms of the types of their parts. A *proof tree* results when typing rules are employed in sequence to determine the type of an entire **svexp** program. The inference algorithm constructs proof trees automatically, using a *unification* algorithm to apply typing rules.

The second pass is similar to the first, except that full-fledged *types* replace axis types. Types encode dimension functions in addition to axis sets and element do-

[1]Avoiding dynamic type errors is thus equivalent to the halting problem.

mains. A small language of arithmetic expressions encodes dimensions. Construction of proof trees relies on axis types computed in pass one to choose typing rules and "guess" the types of recursive functions before their types are fully known. Type unification generates a set of *dimension equality constraints* as a side effect. A program for which type inference succeeds and wherein equality constraints are satisfied is certain to produce an s-val result or diverge. Moreover, the algorithm produces a unique "most general" typing. Roughly speaking, this means that the types inferred by the algorithm encode all possible typings that can be achieved using the typing rules.

4.2 The Axis Type Language

We begin with a suitable syntactic representation of axis sets called an *axis map*:

$$\begin{aligned} \langle map \rangle &::= \emptyset \mid [\langle adim \rangle / \langle axis\ number \rangle]\langle map \rangle \mid \langle map\ variable \rangle \langle map \rangle \\ \langle adim \rangle &::= \bullet \mid \langle adim\ variable \rangle \mid \circ \end{aligned} \tag{4.1}$$

An axis map (or adim) with no variables is called a *ground map* (or *ground adim*). The name "adim" is a contraction of *abstract dimension.* In the second pass, we will replace adim with *dim*, the language of dimension arithmetic mentioned in Section 4.1.2. Here we are concerned only with ascertaining whether or not a dimension will be needed at all.

To motivate the syntax of axis maps, we state a key property in advance of the formalities. The interpretation of a map M as an axis set A satisfies the following.

(a) $[\bullet/a]$ occurs anywhere in M only if $a \in A$.

(b) $[\circ/a]$ occurs anywhere in M only if $a \notin A$.

Thus, the association of $\bullet$ with axis a in the map pair $[\cdot/\cdot]$ represents certainty that $a \in A$. Similarly, $[\circ/a]$ in the map represents certainty that axis $a \notin A$. It follows that an axis map with both $[\bullet/a]$ and $[\circ/a]$ has no interpretation as an axis set. This is an *invalid* map. All others are *valid.* Since the pairs containing $\bullet$ and $\circ$ denote axis set membership or the lack of it, their order is not important, and many maps may have the same interpretation.

The notion of maps with the same interpretation is important, and carries forward into axis types. As we detail the interpretations of axis types, we will simultaneously describe the equivalence relation $\equiv$, consisting of all pairs of maps and axis types with the same interpretation.

The formal interpretation of a valid ground map M is simply

$$\{a \mid [\bullet/a] \text{ occurs in } M\}.$$

This is sufficient to characterize $\equiv$ for valid ground maps by set equality. For example,

$$[\bullet/a]\,[\circ/a_2]\emptyset \equiv [\circ/a_1]\,[\bullet/a]\emptyset$$

Non-ground maps are interpreted in the context where they are employed; this is described below.

We next need a language of *base types* to stand for the base domains. This consists of a set of *base type symbols*, e.g. **Int** and **Bool**, and also *base type variables.*

We arrange two maps and a base type in a lexical triple, called an *extent type*, to represent an entire extent. This is written

$$\langle\!\langle M_A,\ M_C,\ \tau \rangle\!\rangle$$

where M_A is the outer axis set map, M_C is the inner, and τ is the base type. We interpret *valid* ground extent types, those with two valid ground maps, as the set of s-vals they describe plus bottom.

Finally, a general *axis type* is either an extent type, or a *function type* of the form:

$$\langle\!\langle M_{C1},\ \tau_1 \rangle\!\rangle, \ldots, \langle\!\langle M_{Cn},\ \tau_n \rangle\!\rangle \rightarrow \langle\!\langle M,\ \tau \rangle\!\rangle$$

Since user functions operate on matrices over base domains rather than s-vals, the extents here have no outer map. We interpret valid ground function types, those containing only valid ground extent types, as the obvious function space including bottom. A *ground axis type* has only ground maps and base types.

A *substitution* is a function taking map, adim, and base type variables to maps, adims, and base types respectively. We will also apply substitutions to complete maps and axis types, where we assume they return the map or type with variables substituted, but otherwise unchanged. Thus the trivial substitution is the identity.

We interpret an extent type T with variables as the union of all interpretations induced by all substitutions taking T to a valid ground type. Likewise, if $T_1, \ldots, T_n \to T$ is a function type with variables, its interpretation is the space of all continuous functions f (including $\perp$) such that for all substitutions S, if E_k is in the interpretation of $(S\,T_k)$ for $1 \leq k \leq n$, then $(f\,E_1\,\ldots\,E_n)$ is in the interpretation of $(S\,T)$. We write $V : T$ to assert that value V is in the interpretation of T.

The relation $\equiv$ extends to types and substitutions as well. For types T_1 and T_2, we have $T_1 \equiv T_2$ if and only if for any substitution S taking these types to ground we have $(S\,T_1) \equiv (S\,T_2)$. It is easy to show by induction on the structure of axis types that this extension is the same as one we might have given inductively on the structure of axis types themselves:

Theorem 4.1 *For axis types T_1 and T_2, $T_1 \equiv T_2$ if and only if one of the following hold:*

(a) $T_1 = \langle\!\langle M_{A1},\ M_{C1},\ \tau_1 \rangle\!\rangle$ and $T_2 = \langle\!\langle M_{A2},\ M_{C2},\ \tau_2 \rangle\!\rangle$, where $M_{A1} \equiv M_{A2}$ and $M_{C1} \equiv M_{C2}$ and $\tau_1 = \tau_2$,

(b) T_1 and T_2 are function types of the same arity, wherein corresponding extent types are equivalent with respect to $\equiv$. $\square$

Intuitively, this is good because we can focus on the equivalence of maps, presuming that full axis types will take care of themselves in proofs of soundness and completeness. Finally, two substitutions S and S' are equivalent with respect to $\equiv$ if they have the same domain, and for any element x in the domain, $(S\,x) \equiv (S'\,x)$.

The point of developing the relation $\equiv$ is to extend the normal concept of syntactic substitution as follows:

Definition 4.2 *Let S be a substitution. Then T' is a* substitution instance of T under S *if* $(S\ T) \equiv T'$. $\square$

Thus the substitution instances of any axis type are in general a superset of the axis types we can get by simple syntactic substitution. For instance, if T is just a variable x and S takes s to $[\bullet/a_1]\emptyset$, then $[\bullet/a_1]\emptyset$ is a substitution instance of x under S, but so are all equivalent types with respect to $\equiv$, e.g. $[\bullet/a_1]\,[\circ/a_2]\emptyset$.

This completes the discussion of the syntax and interpretation of axis types, and we are ready to state the property of axis types that justifies all the effort:

> Values from $\mathbf{W}_t$ (the type error values) are not in the interpretation of any valid axis type.

We exploit this by seeking an algorithm to associate a valid axis type with each **svexp** subterm in a program such that *all possible* values the subterm can take are in the interpretation of its type. Then, whenever the algorithm succeeds, the property above assures us that the program cannot produce a type error. This is exactly the *semantic soundness* property of conventional typing rules. Unfortunately, there exists no such algorithm in our case, because the language of axis maps is insufficient to code the dimension constraints imposed by the semantic equations for Join and Cond. However, we *can* achieve soundness in a weaker sense, wherein all possible values a subterm can take are either in the interpretation of its type or are $typerr_2$. We take care of $typerr_2$ values in the second inference pass, where general *maps* replace axis maps.

4.3 Axis Set Inference Rules

Our interpretation of types taken together with the theorems of Chapter 3 give rise to succinct axis set inference rules that lead to the desired algorithm.

4.3.1 A look at Typ

Consider the primitive "Typ $c\,a$". Equation (3.1) shows that the operand *must* have inner axis c; it may or may not have outer axis a. From the definitions above, one type that encodes all such s-vals (as well as $\perp$) is

$$T_1 = \langle\!\langle\, [d/a]m,\ \ [\bullet/c]n,\ b \rangle\!\rangle \tag{4.2}$$

Here we adopt the convention that letters $m, n, \ldots$ are map variables, $d, e, \ldots$ are adim variables, and b is a base type variable.

Now consider s-vals that can result from a Typ application. Theorem 3.8 shows that inner axis c surely does *not* occur in the result and that outer axis a surely *does*. Furthermore, the axis type of inner matrix elements in the result is the same as in the input. Hence, in the context of (4.2), we can code all possible result extents with

$$T_2 = \langle\!\langle\, [\bullet/a]m,\ \ [\circ/c]n,\ b \rangle\!\rangle \tag{4.3}$$

With this derivation, we have shown the following:

Theorem 4.3 *For any substitution S, if $V : (S\,T_1)$, then $\mathcal{I}[\![\text{Typ } c\ a]\!]\ V : (S\ T_2)$.* $\Box$

The reasoning behind (4.2) and (4.3), hence Theorem 4.3, includes semantic values. However, for compilation we need purely syntactic rules that *assign* a particular axis type to each **svexp** subterm. Since subterms may include free variables, any such rule must be given with respect to a *type environment*, denoted here with Γ. This is just a map from **svexp** variables (including function variables) to axis types.[2]

[2]Do not confuse type environments with substitutions. The former "store" the

A term E that can be assigned axis type T by applying the rules with respect to environment Γ is said to *have* axis type T in Γ, written $\Gamma \vdash E : T$.[3] Here is the rule for Typ:

$$\frac{\Gamma \vdash E : \langle\!\langle\, [d/a]m,\ \ [\bullet/c]n,\ b \rangle\!\rangle}{\Gamma \vdash \mathrm{Typ}\, c\, a\, E : \langle\!\langle\, [\bullet/a]m,\ \ [\circ/c]n,\ b \rangle\!\rangle}\quad \mathit{TYP} \tag{4.4}$$

The top of the rule is its *antecedent*; the bottom is its *consequent*. We read it as follows:

> For all substitutions S, if E has an axis type that is a substitution instance of $\langle\!\langle\, [d/a]m,\ \ [\bullet/c]n,\ b \rangle\!\rangle$ under S in Γ then Typ $c\ a\ E$ has (in Γ) all types T such that
>
> $$T \equiv (S\ \langle\!\langle\, [\bullet/a]m,\ \ [\circ/c]n,\ b \rangle\!\rangle)$$

An essential quality of this rule is that if the semantic value of E is in the interpretation of the axis type $(S\ \langle\!\langle\, [d/a]m,\ \ [\bullet/c]n,\ b \rangle\!\rangle)$ for some substitution S, then the semantic value of Typ $c\ a\ E$ is in the interpretation of $(S\ \langle\!\langle\, [\bullet/a]m,\ \ [\circ/c]n,\ b \rangle\!\rangle)$. This property is of course *semantic soundness* for just this rule. Hence we should write it carefully:

Theorem 4.4 (Semantic soundness of Typ) *Let ϕ and β be function and bound variable environments and let Γ be a type environment wherein E has axis type T and* Typ $c\ a\ E$ *has type T' by rule TYP. Then*

$$\mathcal{E}[\![E]\!]\,\phi\,\beta : T \quad \text{implies} \quad \mathcal{E}[\![\mathrm{Typ}\ c\ a\ E]\!]\,\phi\,\beta : T'$$

---

types of **svexp** variables, allowing the typing rules given shortly to insist that all occurences of the same **svexp** variable have the same type. On the other hand, substitutions store bindings for *type* variables.

[3] We are invoking the slick (and occasionally abused) convention of the literature of overloading the colon. To recap our usage, $\vdash E : T$, where E is a **svexp** *subterm*, means that E may be proven with type inference rules to have an axis type of the form T. Conversely, $V : T$ where V is a *semantic value* asserts that V is in the interpretation of T.

Table 4.1: Primitive typing rules of unqualified soundness.

$$\frac{\Gamma \vdash E : \langle\!\langle\, [d/a]m,\ \ [\bullet/c]n,\ b \rangle\!\rangle}{\Gamma \vdash \mathrm{Typ}\, c\, a\, E : \langle\!\langle\, [\bullet/a]m,\ \ [\circ/c]n,\ b \rangle\!\rangle} \qquad \mathit{TYP}$$

$$\frac{\Gamma \vdash E : \langle\!\langle\, [\bullet/a]m,\ \ [\circ/c]n,\ b \rangle\!\rangle}{\Gamma \vdash \mathrm{Proj}\ a\ c\, E : \langle\!\langle\, [\circ/a]m,\ \ [\bullet/c]n,\ b \rangle\!\rangle} \qquad \mathit{PROJ}$$

$$\frac{\Gamma \vdash E : \langle\!\langle m,\ \ [\bullet/c]n,\ b \rangle\!\rangle}{\Gamma \vdash \mathrm{Extr}_0\ c\, E : \langle\!\langle m,\ \ [\circ/c]n,\ b \rangle\!\rangle} \qquad \mathit{EXTR0}$$

$$\frac{\Gamma \vdash E : \langle\!\langle m,\ \ [\bullet/c]n,\ b \rangle\!\rangle}{\Gamma \vdash \mathrm{Extr}_1\ c\, E : \langle\!\langle m,\ \ [\bullet/c]n,\ b \rangle\!\rangle} \qquad \mathit{EXTR1}$$

$$\frac{\Gamma \vdash E : \langle\!\langle m,\ \ [\bullet/c]\emptyset,\ \mathbf{Int} \rangle\!\rangle}{\Gamma \vdash \mathrm{Red}+\ \ c\, E : \langle\!\langle m,\ \emptyset,\ \mathbf{Int} \rangle\!\rangle} \qquad \mathit{RED+}$$

$$\frac{\Gamma \vdash E_1 : \langle\!\langle m_1,\ \emptyset,\ \mathbf{Int} \rangle\!\rangle,\ E_2 : \langle\!\langle m_2,\ \emptyset,\ \mathbf{Int} \rangle\!\rangle}{\Gamma \vdash\ +\ E_1\, E_2 : \langle\!\langle m_1 m_2,\ \emptyset,\ \mathbf{Int} \rangle\!\rangle} \qquad \mathit{BIN+}$$

Proof. Immediate from Theorem 4.3 and Equation 3.11, and the facts that $\perp$ has all types and $typerr_1$ has none. □

4.3.2 Primitive typing rules of unqualified soundness

The procedure that led to a sound inference rule for Typ works as well for most of the other **svexp** primitives. These are given in Table 4.1, which is titled *unqualified* because the rules require no other considerations for semantic soundness. *PROJ*, *EXTR0*, *EXTR1*, are consequences of (3.21), (3.22), and (3.23) respectively, as well as Theorem 3.8. *RED+* is due to (3.24). It forces the input to be an inner vector of integers along axis c with a map that is a single pair and **Int** for the argument base type. The inner map for the output is empty, as the reduced output is inner-scalar.

Table 4.2: Axis typing rules of qualified soundness.

$$\frac{\Gamma \vdash E_1 : \langle\!\langle m_1,\ [d_1/c]n,\ b \rangle\!\rangle,\ E_2 : \langle\!\langle m_2,\ [d_2/c]n,\ b \rangle\!\rangle}{\Gamma \vdash \text{Join } c\ E_1\ E_2 : \langle\!\langle m_1\, m_2,\ [\bullet/c]n,\ b \rangle\!\rangle} \quad \mathit{JOIN}$$

$$\frac{\Gamma \vdash B : \langle\!\langle m_B,\ \emptyset,\ \mathbf{Bool} \rangle\!\rangle,\ T : \langle\!\langle m_T,\ n,\ b \rangle\!\rangle,\ F : \langle\!\langle m_F,\ n,\ b \rangle\!\rangle}{\Gamma \vdash \text{Cond } B\ T\ F : \langle\!\langle m_B\, m_T\, m_F,\ n,\ b \rangle\!\rangle} \quad \mathit{COND}$$

4.3.3 Primitive axis typing rules of qualified soundness

The remaining primitives have semantics that restrict the inner dimensions of their arguments. For "Join c", the inner dimensions of the arguments must match except along axis c to satisfy *conformable*$_2$ (Definition 3.9 and Equation (3.25)). For "Cond" the true and false branch values must match exactly in their inner dimensions to satisfy Q_2 in Equation (3.30). We cannot code these constraints in the language of axis types because they involve equality of dimensions that are not expressed at all by axis types. As mentioned above, we solve this problem by deferring it to the second pass. Otherwise, the axis typing rules for Join and Cond given in Table 4.2 are derived in the same manner as *TYP*. The only important difference is that *typerr*$_2$ is a possible result value. *JOIN* encodes the axis set constraints of *conformable* given in Equation (3.25) and Theorem 3.10 by insisting that the inner axis sets of both arguments and the result match except for axis c. The reader should recall that axis c itself may properly be present or absent as the inputs are promoted. This is reflected by pairing c with different adim variables in each input. On the other hand, axis c is certainly present in the output, hence c is paired with $\bullet$ in the consequent. The union of the inputs' outer axis sets in the result is coded by concatenating the inputs' outer maps in the rule. Similar reasoning gives us *COND* from (3.30).

Table 4.3: Type rules (axioms) for constants and variables.

$$\Gamma \vdash \begin{bmatrix} x_{0,0} & \cdots & x_{0,q-1} \\ \vdots & \ddots & \vdots \\ x_{p,0} & \cdots & x_{p-1,q-1} \end{bmatrix} : \langle\!\langle \emptyset,\, [\bullet/0]\,[\bullet/1]\emptyset, \mathbf{B}_k \rangle\!\rangle \qquad \textit{CONST}$$

$$\Gamma \vdash v : (\Gamma\, v) \quad \text{(bound or function variable expression)} \qquad \textit{VAR}$$

4.3.4 Other program structures

It is not hard to write rules for the rest of **svexp** that make sense in terms of our understanding of user function application, environments, etc. There is, however, no background like Theorem 3.8 to make soundness clear for these constructs. Happily, they succumb to structural induction of the same sort used to prove soundness of more conventional languages. Thus, the hard rules are done, but we are obliged to be careful enough with the rest to achieve this next milestone:

Theorem 4.5 (Qualified semantic soundness of axis typing) *If P is a* **svexp** *program, then $\emptyset \vdash P : T$ implies that either $\mathcal{P}[\![P]\!] : T$, or else $\mathcal{P}[\![P]\!] = typerr_2$.*

The remaining type rules are the axioms that form the leaves of derivation trees, given in Table 4.3, and the rules for typing function variables and applications, Table 4.4. The former are self-explanatory; the latter are straightforward, but we comment briefly. *ABS* has a conjunction of requisites:

(a) The rule first requires that the function variables defined before f_k in syntactic order have (function) types F_1 through F_{k-1} respectively.

(b) It next augments the type environment. It binds f_k to F_k (with a *generic* tag explained below) for $0 \leq k < i$, and also f_k to a "guessed" function type, and finally f_k's arguments to extents that match the function argument types, but with empty outer axis sets. In this environment, E is required to have some extent with an empty outer axis set.

The F_k in (a) must be proved with earlier applications of *ABS*. The empty outer axis sets derive from Equation 3.15, where they correspond to the results of S. Similarly, the empty outer axis set of E derives from the check in Equation 3.32. The *generic* tag is a bit of additional type syntax that distinguishes calls to previously defined functions from recursive calls. In the former case, we have already proved the final type of the called function and so can freely instantiate its type by substitution (rule *INST*). In the latter, we cannot allow general instantiation without introducing infinite types, and so have made a common restriction, allowing no instantiations at all.

Formally, the type $generic(F)$ has the same interpretation as F. However, all its variables are immune from substitution; i.e. for all substitutions S, the interpretation of $(S\, generic(F))$ is the same as that of $generic(F)$ and F.

The last necessity is a systematic treatment of environments. Values given by $\mathcal{E}$ and $\mathcal{D}$ depend on bound and function variable environments, and typing rules depend on type environments. Intuitively, we are interested only in cases where variable and type environments "match." Formally, we say β *respects* Γ if for all variables x bound in β, x is also bound in Γ and $(\beta\, x) : (\Gamma\, x)$. With this, we prove the qualified soundness of $\mathcal{E}$ and $\mathcal{D}$.

Lemma 4.6 *Let* $V' = \mathcal{E}[\![E']\!]\phi\beta$ *and suppose* ϕ *and* β *respect* Γ. *Then* $\Gamma \vdash E' : T'$ *implies* $V' : T'$ *or else* $V' = typerr_2$.

Proof. By induction on the structure of E'. Possible cases are as follows.

E' *is* Typ $c\, a\, E$. Suppose $\Gamma \vdash E' : T'$. Since this can only be proven with typing rule *TYP*, there must exist T such that $\Gamma \vdash E : T$. From the hypothesis we then

Table 4.4: Axis type rules for general program structures.

$$\frac{\begin{array}{l}\emptyset \vdash f_1 : F_1, \ldots\ \emptyset \vdash f_{k-1} : F_{k-1}, \\ [\mathit{generic}(F_1)/f_1]\ \ldots\ [\mathit{generic}(F_{k-1})/f_{k-1}] \\ \qquad [\langle\!\langle n_1, b_1\rangle\!\rangle, \ldots, \langle\!\langle n_n, b_n\rangle\!\rangle \ \rightarrow\ \langle\!\langle n, b\rangle\!\rangle / f_k] \\ \qquad [\langle\!\langle \emptyset, n_1, b_1\rangle\!\rangle / x_1]\ \ldots\ [\langle\!\langle \emptyset, n_n, b_n\rangle\!\rangle / x_n] \ \vdash\ E : \langle\!\langle \emptyset, n, b\rangle\!\rangle\end{array}}{\emptyset \vdash f_k : \langle\!\langle n_1, b_1\rangle\!\rangle, \ldots, \langle\!\langle n_n, b_n\rangle\!\rangle \ \rightarrow\ \langle\!\langle n, b\rangle\!\rangle \text{ (for } f_k \text{ in } f_k\, x_1 \ldots x_n \equiv E)} \qquad \mathit{ABS}$$

$$\frac{\Gamma \vdash f\colon \mathit{generic}(F)}{\Gamma \vdash f\colon F'} \quad \text{where } F' \text{ is a substitution instance of } F \qquad \mathit{INST}$$

$$\frac{\begin{array}{l}\Gamma \vdash f\colon \langle\!\langle n_1, b_1\rangle\!\rangle, \ldots, \langle\!\langle n_n, b_n\rangle\!\rangle \ \rightarrow\ \langle\!\langle n, b\rangle\!\rangle, \\ \qquad E_1 : \langle\!\langle m_1, n_1, b_1\rangle\!\rangle, \ldots, E_n : \langle\!\langle m_n, n_n, b_n\rangle\!\rangle\end{array}}{\Gamma \vdash f\, E_1 \ldots E_n : \langle\!\langle m_1\ \ldots\ m_n, n, b\rangle\!\rangle} \qquad \mathit{APP}$$

$$\frac{\begin{array}{l}\emptyset \vdash f_1 : F_1, \ldots, f_n : F_n, \\ [\mathit{generic}(F_1)/f_1] \cdots [\mathit{generic}(F_n)/f_n] \vdash E : \langle\!\langle \emptyset, n, b\rangle\!\rangle\end{array}}{\emptyset \vdash \text{def } f_1 \cdots f_n \cdots \text{in } E : \langle\!\langle \emptyset, n, b\rangle\!\rangle} \qquad \mathit{PROG}$$

have $\mathcal{E}[\![E]\!]\phi\beta : T$, and finally $V' : T'$ by Theorem 4.3.

E' is Proj $a\,c\,E$, Extr$_i$ $c\,E$, Red+ $c\,E$, *or* + $E_1\,E_2$. Follows from unqualified soundness of these primitives and typing rules as in the former case. The equations corresponding to (3.20) in the proof of Theorem 4.3 are (3.21), (3.22), (3.23), (3.24), and (3.26) respectively. The typing rules corresponding to *TYP* are *PROJ*, *EXTR0* and *EXTR1*, *RED+*, and *BIN+* respectively.

E' is Join $c\,E_1\,E_2$ *or* Cond $E_B\,E_T\,E_F$. Proofs are similar to the former case, except the semantic equation may also produce $typerr_2$, whereupon we have proven the *or else* clause of the lemma. See equations (3.25) and (3.27) and typing rules *JOIN* and *COND*.

E' is variable v. Then $V' : T'$, by one of three cases. If v is a bound variable, we have $T' = (\Gamma\,v)$ from axiom *VAR*. Then $V' : T'$ by (3.13) and the fact β respects Γ. Otherwise v is a function variable. Assume without loss of generality that T' does not have a generic tag. Then T' was derived with *VAR* (the second case), or else with *VAR* followed by *INST* (the third). In the second case, $V' : T'$ from (3.14) and because ϕ respects Γ. In the third case, $(\Gamma\,v) = generic(F)$ and T' is a substitution instance of F. Here $V' : T'$ as above with this additional fact concerning types: If $f : F$, then $f : (S\,F)$ for all substitutions S.

E' is $f\,E_1 \cdots E_n$. From rule *APP*, we know there exists substitution S such that

$$\begin{aligned}&T' = (S\,\langle\!\langle m_1 \cdots m_n, n, b\rangle\!\rangle) \quad \text{and}\\ &\Gamma \vdash E_k : (S\,\langle\!\langle m_k, n_k, b_k\rangle\!\rangle),\ 1 \le k \le n \quad \text{and}\\ &\Gamma \vdash f : (S\,\langle\!\langle n_1, b_1\rangle\!\rangle, \ldots, \langle\!\langle n_n, b_n\rangle\!\rangle \rightarrow \langle\!\langle n, b\rangle\!\rangle)\end{aligned}$$

Thus $\mathcal{E}[\![E_k]\!]\phi\beta : (S\,\langle\!\langle m_k, n_k, b_k\rangle\!\rangle)$ and $\mathcal{E}[\![f]\!]\,\phi\,\beta : S\,(\langle\!\langle n_1, b_1\rangle\!\rangle, \ldots, \langle\!\langle n_n, b_n\rangle\!\rangle \rightarrow \langle\!\langle n, b\rangle\!\rangle)$ by hypothesis. From semantic equation (3.12) and Theorem 3.16 we conclude $V' : T'$ as desired.

This completes the cases. □

Lemma 4.7 *Suppose* $\emptyset \vdash f_i : T_1, \ldots, T_n \to T$. *Let* $[g_i/f_i] = \mathcal{D}[\![f_i\, x_1 \cdots x_n \equiv E]\!]\phi$ *as in Equation (3.15), where* ϕ *respects* $\Gamma = [\mathit{generic}(F_1)/f_1] \cdots [\mathit{generic}(F_n)/f_n]$ *with* $\emptyset \vdash f_k : F_k,\ 1 \leq k < i$. *Then* $\emptyset \vdash f_i : T_1, \ldots, T_n \to T$ *implies* $g_i : T_1, \ldots, T_n \to T$, *or else* g_i *returns* typerr_2 *on some input.*

Proof. Let $M_k,\ 1 \leq k \leq n$ be matrices. By the interpretation of function types, we must show that for all substitutions S and input matrices M_k such that M_k has type $(S\,T_k)$ for $1 \leq k \leq n$, we have $g_i\, M_1 \cdots M_n : (S\,T)$. From the typing rule for f_i, we know

$$\Gamma\,[T_1/x_1] \cdots [T_n/x_n]\,[T_1, \ldots, T_n \to T)/f_i] \vdash E : T$$

A simple induction proves that if $\Gamma \vdash E : T$, then $(S\,\Gamma) \vdash E : (S\,T)$. Hence,

$$(S\,\Gamma\,[T_1/x_1] \cdots [T_n/x_n]\,[T_1, \ldots, T_n \to T/f_i]) \vdash E : (S\,T)$$

Since Γ contains only generic types, it is not affected by substitution, and we have

$$\Gamma\,[(S\,T_1)/x_1] \cdots [(S\,T_n)/x_n]\,[(S\,T_1, \ldots, T_n \to T)/f_i]) \vdash E : (S\,T)$$

Let $V = \mathcal{E}[\![E]\!]\,\phi\,[(S\,T_1, \ldots, T_n \to T)/f_i]\,\bot\,[\lambda i\,.\,M_1/x_1] \cdots [\lambda i\,.\,M_n/x_n]$. We have by Lemma 4.6 that for all substitutions S, if $M_k : (S\,M_k),\ 1 \leq k \leq n$, then $(V\,\mathbf{0}) : (S\,T)$ or else $V = \mathit{typerr}_2$. This yields the desired result by a conventional continuity argument and the fact that if $V : T$ and $V' \sqsubseteq V$, then $V' : T$. □

Proof of Theorem 4.5 for soundness of entire programs follows similarly from these two lemmas and typing rule *PROG*. □

4.3.5 Automatic axis type derivations

We now have a set of rules that allow us to derive axis types for a program by chaining together rule applications for each program substructure. Since some of

$$\Gamma = [\langle\!\langle\, [\bullet/0]\emptyset, \mathbf{Int} \rangle\!\rangle/x]\,[\langle\!\langle\, [\bullet/1]\emptyset, \mathbf{Int} \rangle\!\rangle/y]\,[\langle\!\langle\, [\bullet/0]\emptyset, \mathbf{Int} \rangle\!\rangle, \langle\!\langle\, [\bullet/1]\emptyset, \mathbf{Int} \rangle\!\rangle \rightarrow \langle\!\langle\, [\bullet/2]\emptyset, \mathbf{Int} \rangle\!\rangle/f]$$

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash x : \langle\!\langle\, [\circ/9]\emptyset,\ [\bullet/0]\emptyset, \mathbf{Int} \rangle\!\rangle}{\Gamma \vdash \mathrm{Typ}\,0\,9\,x : \langle\!\langle\, [\bullet/9]\emptyset,\ [\circ/0]\emptyset, \mathbf{Int} \rangle\!\rangle}\,TYP \qquad \dfrac{\Gamma \vdash y : \langle\!\langle\, [\circ/9]\emptyset,\ [\bullet/1]\emptyset, \mathbf{Int} \rangle\!\rangle}{\Gamma \vdash \mathrm{Typ}\,1\,9\,y : \langle\!\langle\, [\bullet/9]\emptyset,\ [\circ/1]\emptyset, \mathbf{Int} \rangle\!\rangle}\,TYP}{\dfrac{\Gamma \vdash\ +(\mathrm{Typ}\,0\,9\,x)(\mathrm{Typ}\,1\,9\,y) : \langle\!\langle\, [\bullet/9]\emptyset,\ [\circ/2]\emptyset, \mathbf{Int} \rangle\!\rangle}{\Gamma \vdash \mathrm{Proj}\,9\,0(\,+(\mathrm{Typ}\,0\,9\,x)(\mathrm{Typ}\,1\,9\,y)) : \langle\!\langle\, [\circ/9]\emptyset,\ [\bullet/2]\emptyset, \mathbf{Int} \rangle\!\rangle}\,PROJ}\,BIN{+}}{\emptyset \vdash f : \langle\!\langle\, [\bullet/0]\emptyset, \mathbf{Int} \rangle\!\rangle, \langle\!\langle\, [\bullet/1]\emptyset, \mathbf{Int} \rangle\!\rangle \rightarrow \langle\!\langle\, [\bullet/2]\emptyset, \mathbf{Int} \rangle\!\rangle}\,ABS$$

Figure 4.1: Deriving an axis type.

the rules have conjunctive antecedents, chaining forms a *derivation tree*. Consider a derivation of the type of f in following definition:

$$f\ x\ y \equiv \mathrm{Proj}\ 9\ 2\ \Big(\ +\ (\mathrm{Typ}\ 0\ 9\ x)\ (\mathrm{Typ}\ 1\ 9\ y)\ \Big)$$

We derive $\emptyset \vdash f : \langle\!\langle\, [\bullet/0]\emptyset, \mathbf{Int} \rangle\!\rangle, \langle\!\langle\, [\bullet/1]\emptyset, \mathbf{Int} \rangle\!\rangle \rightarrow \langle\!\langle\, [\bullet/2]\emptyset, \mathbf{Int} \rangle\!\rangle$ as shown in Figure 4.1.

The tree reads top to bottom as a proof of the type of f from axioms, constructed by hypothesizing types for x and y and chaining forward to prove the type of f. Alternatively, we can write the term to be typed at the bottom of the page and apply rules backward by fitting a rule consequent to the syntax for the subterm to be typed next.

The important question is this: How can a machine employ the rules to find an axis type for any program P, deriving $\emptyset \vdash P : T$? Related tasks are very hard—a large community studies *automatic theorem proving*, where the space of possible proofs is usually exponential or worse in the size of the theorem. In one sense, our task is even harder. We would like not only to get *some* derivation, but to derive, at every subterm, a *most general* or *principal* axis type, a type of which every other valid type is a substitution instance. This makes the algorithm *complete*—there are no type derivations under the rules that it will miss. Of course, we will also be concerned for the *syntactic soundness* of the algorithm. When it returns a type, it had better correspond to a derivation tree. Fortunately, our rules have a special structure that makes derivations easy to come by. At most one rule applies to any node in a proof tree; the derivation is precisely directed by **svexp** syntax. General theorem provers do not have this luxury. The following sections are devoted to constructing an efficient algorithm that exploits this structure to achieve sound and complete type inference.

4.4 Axis Type Unification

In constructing the derivation tree by hand above, there is never any doubt about the rule to apply next. The **svexp** syntax directs the derivation. The only ingenuity required is, for each rule, to deduce a substitution that makes the rule antecedent equivalent with respect to $\equiv$ to a previously derived type. As long as we can find such a substitution in each case, the derivation succeeds.

This section concerns a subroutine that replaces our ingenuity, finding the required substitution whenever one exists. There are two phases. In the first, we construct the entire proof tree from syntax without regard to substitutions, noting for each rule application where $\equiv$ must hold for the derivation to succeed. These notes form a set of *axis type equations.* Type inference is thus reduced to solving such a set of equations, which is the purpose of the second phase. This division of labor simplifies the soundness and completeness proofs. It is adapted from the technique of [Wan87].

Specifically, we say Q is *a set of type equations* if it is a set composed of elements of the form $L_i \stackrel{?}{=} R_i$, where L_i and R_i are axis types. We say $S \models Q$ iff S is a substitution, Q is a set of type equations, and for all equations $L_i \stackrel{?}{=} R_i$ in Q, we have $(S\, L_i) \equiv (S\, R_i)$. In other words, S takes both sides of each equation to types with the same interpretation. This is a generalization of the classical unification of [Rob65], which is adequate for inferring types in ML.

Definition 4.8 *Given substitutions S and S', S is* more general than *S', written $S \succeq S'$, iff there exists a third substitution T such that $S' \equiv T \circ S$. A substitution S is a* unifier *for the set of type equations Q iff $S \models Q$. It is a* most general unifier *iff for all other substitutions S' such that $S' \models Q$, we have $S \succeq S'$.* □

The usual purely syntactic definition of $\succeq$ is inadequate here because many axis types have the same interpretation. For example, consider the solution of $Q = \{m \stackrel{?}{=} \emptyset\}$. One most general unifier is the singleton map $[\emptyset/m]$. However, $[\,[\circ/0]\emptyset/m]$ serves just as well because the interpretation of m after one or the other unifier is applied is the same. The essence of a most general unifier, then, is that it provides the largest possible interpretation of all the axis types appearing in the equation set. This *interpretation* must be unique; otherwise there is no most general unifier. However, as in the case just shown, the *unifier that induces it* is usually not unique.

The skeleton of the unification algorithm is given in Figure 4.2. It processes equations by cases based on their structure as described below. Processing builds up the substitution S, which eventually is returned as the answer. An equation that is eligible for processing is called *active.* Equations can have several histories; they can be:

- Processed once and discarded from Q.
- Processed once and replaced in Q with fresh, active equations.
- Modified and placed (inactive) back in Q, with a chance that some future processing may cause reactivation.

Algorithm: Axis type unification

Input: A set Q_0 of axis type equations.

Outputs: Either the special value *fail* (no unifier exists) or a most general unifier for Q_0.

Method: Let S be the identity substitution;
Let $Q = Q_0$;
While there is an active equation $e = (L_i \stackrel{?}{=} R_i)$ in Q
  {*Invariant*}
  Remove e from Q;
  Process e as described below;
    (possibly adding to Q, extending S, or returning *fail*)
  {*Progress criterion*}
End while;
{*Invariant*}
Return S.

Figure 4.2: The axis type unification algorithm.

The following help to show correctness and termination:

Invariant. (a) For any substitution S' such that $S' \models Q_0$, we have $S \succeq S'$. In particular, we can exhibit a unifier U of Q such that $U \circ S \equiv S'$. (b) For any unifier U of Q, we have $U \circ S \models Q_0$.

Progress criterion. Make a lexicographic count of three quantities in the following priority order (highest first):

- Occurrences of map variables in equations active since the beginning, the *always-active equations.*
- The number of distinct dimension variables in Q.
- The number of other symbols in active equations.

The criterion itself is that the count decreases at each iteration.

The invariant states (a) that S is more general than any unifier of Q_0—including the most general one—and (b) that S monotonically approaches a unifier of Q_0 as Q gets less active. The progress criterion ensures Q must finally be entirely inactive. At this point, we shall see, any substitution unifies Q, including the identity. Thus (b) becomes $S \models Q_0$; i.e., S is a unifier, and (a) makes it most general as desired. Moreover, the contrapositive reading of (a) ensures that if there is no unifier for Q, then there is no unifier for Q_0, and *fail* is the correct answer.

It remains to give the processing cases for equations e and show that the invariant and progress criterion are satisfied at each iteration. Initially, $U = S'$ satisfies invariant clause (a), and (b) holds trivially. To ease presentation, the cases are written presuming that earlier ones do not apply.

Function types. If either L_i or R_i is a function type and the other is not, there is clearly no unifier for Q, so return *fail.*[4] Otherwise, if $L_i = E_1, \ldots, E_n \rightarrow E$ and $R_i = E'_1, \ldots E'_n \rightarrow E'$ (i.e., both are function types), then add to Q the equations $E_1 \stackrel{?}{=} E'_1, \ldots, E_n \stackrel{?}{=} E'_n, E \stackrel{?}{=} E'$.

We now argue that invariant (a) still holds. For any S', let U be the unifier that satisfied the invariant previous to this processing step. Then $U \models Q$ even after processing by Theorem 4.1. Thus $U \circ S \equiv S'$ because we haven't changed S.

For invariant (b), suppose U unifies the new equations. Then it also unified the old ones by Theorem 4.1. It follows that since we previously had $U \circ S \models Q_0$, and S has not changed, we still have it.

This argument is simple, but representative of other cases to follow where the interpretation of types guides us directly to a unifying step that does not modify S. As such, we refer to it as the *constant S argument.*

Progress is assured because a "$\rightarrow$" is eliminated from the symbols in the active equations of Q, and the other parts of the lexicographic count are unchanged.

Extent types. If either L_i or R_i is an extent type and the other is not, there is clearly no unifier for Q, so return *fail.* Otherwise, if $L_i = \langle\!\langle M_A, M_C, \tau \rangle\!\rangle$ and $R_i = \langle\!\langle M'_A, M'_C, \tau' \rangle\!\rangle$ (i.e., both are extent types), then add to Q equations $M_A \stackrel{?}{=} M'_A$, $M_C \stackrel{?}{=} M'_C$, and $\tau' \stackrel{?}{=} \tau$. Similarly for the short extents due to function types; just omit M_A and M'_A.

The invariant is preserved by the constant S argument. Progress is assured because symbols "$\langle\!\langle\ \rangle\!\rangle$" are eliminated from the active equations of Q.

Base types. If both L_i and R_i are the same ground type, we are done. If they are different ground types return *fail.* Otherwise one is a variable b and the other is

[4]Recall function arity and argument counts are matched by the syntax of **svexp**, so this mode of failure cannot occur, but we could easily have enforced it here instead.

some base type τ (ground or variable), Replace S with $S[\tau/b]$, and replace b with τ throughout Q.

If we have matching ground base types, then the constant S argument holds.

Otherwise, a variable b was encountered:

To see invariant clause (a), suppose Q_{last} is Q and S_{last} is S prior to this processing step. Then, by assumption, for any S' unifying E_0 there is a unifier U_{last} for Q_{last}, where $U_{last} \circ S_{last} \equiv S'$. Since b appears in Q_{last}, and base type variables are replaced in Q as soon as they are added to S, we know $(S_{last}\, b) = b$. Hence, we must have

$$(U_{last}\, b) \equiv \tau \tag{4.5}$$

or otherwise, U_{last} could not unify E_{last}. We conclude $U_{last} \circ S_{last} \equiv U_{last} \circ S$. Since U_{last} also unifies E, it has all the desired properties of U in the invariant.

For invariant clause (b), suppose $U \models Q$. Then we must have

$$U[\tau/b] \models Q_{last} \tag{4.6}$$

Then, by assumption, $U[\tau/b] \circ S_{last} \models Q_0$. This implies $U \circ S_{last}[\tau/b] \models Q_0$, but $S_{last}[\tau/b]$ is S.

This is the prototypical *augmented S argument.* To employ it elsewhere, we need only give replacements for Equations 4.5 and 4.6 based on the interpretation of the types L_i and R_i at hand.

Progress occurs here because an active equation is deleted from Q, and substitution for base type variables does not change the number of symbols.

If none of the cases thus far apply, then both L_i and R_i are maps.

Adim unification. Unifying maps often involves getting two adims to have the same interpretation. For this, we define a simple subroutine for unification of adims D and D' as follows: If either D or D' is a variable d and the other is some adim A (variable or ground). Replace S with $S[A/d]$ and replace d with A throughout Q.

If both D and D' are ground and match, we are done. If they do not match, return *fail* for the top level unification of Q_0.

This procedure maintains the invariant provided that $(S\,D) \equiv (S\,D')$ is necessary to unify the maps in which D and D' lie. Specifically, if no adim variables are involved, then the invariant is trivially true because nothing has changed. If a substitution is performed, then the augmented S argument applies with Equations 4.5 and 4.6 replaced respectively by $(U_{last}\,d) \equiv A$ and $U\,[A/d] \models Q_{last}$.

Redundant maps. If either L_i or R_i has two pairs $[D/a]$ and $[D'/a]$ (i.e., two pairs in one map have the same axis number), delete the second pair from e and return it to Q. Unify D and D'.

Clearly, the type with the redundant pairs has an interpretation as an axis set only if we can achieve $(S\,D) \equiv (S\,D')$. This is established correctly by adim unification as shown above. Moreover, we are safe in deleting one of the redundant pairs because it does not change the interpretation of the affected type.

To check progress, note that if no variables are substituted, then the elimination of the map pair is sufficient. If an adim variable d was substituted, however, the count of symbols in always-active equations of Q may have increased. This is where the lexicographic order of the progress criterion is important, for since d is now gone from Q, growth in the number of symbols is overbalanced.

No extensible maps. If neither L_i nor R_i has map variables, then consider all the axis numbers that appear therein, and partition them into three sets L, R, and B, namely, those that occur only in L_i, those that occur only in R_i, and those that occur in both. Unify the adims of all pairs with axis numbers in L with $\circ$. Do the same for all pairs with axis numbers in R. Then consider each axis number $a \in B$ and the associated pairs $[D/a]$ from L_i and $[D'/a]$ from R_i. Unify D and D'.

When this case is considered, there are no redundant map pairs in L_i or R_i, so the applications of adim unification are well-defined. Then by the interpretation of maps, we have that $(S\,L_i) \equiv (S\,R_i)$ if and only if their adims are $\equiv$ for axes that appear in pairs of both, and the adims of other pairs are $\circ$. These adim relations are established correctly by adim unification.

Two extensible maps, two map variables. If L_i and R_i each have one map variable, say m and n ($m \neq n$) respectively, then form sets L, R, and B as above, and unify adim pairs for axes in B just as in the previous case. Next, let p be a fresh map variable, and let $\overline{L} = [D_1/a_1] \ldots [D_m/a_m]$ be the pairs associated with the axes in L from L_i and $\overline{R} = [D'_1/\mathrm{a}'_1] \ldots [D'_n/\mathrm{a}'_n]$ be the pairs associated with the axes in R from R_i. Then replace S with

$$S\,[\overline{R}p/m]\,[\overline{L}p/n]$$

and make associated replacements for m and n in Q as well as e. Finally, put e back in Q.

When this case is considered, there are again no redundant map pairs in L_i or R_i, so adim unification on pairs with axis numbers in B is well-defined and is necessary to achieve $(S\,L_i) \equiv (S\,R_i)$. Thus the invariant is preserved to the point where adim unification is complete. Let L'_i and R'_i be L_i and R_i after the replacements are done in adim unification, and let $\overline{B}$ be the pairs associated with the axes in B from, say, L'_i (those in R'_i are identical except possibly for their order).

It remains to show that the chosen substitutions for m and n also preserve the invariant. For this, we give names to the interpretations of various sets of pairs: $L^\bullet$ is the set of axes, a, that definitely *are* in the interpretation of L'_i due to the occurrence of $[\bullet/a]$ in $\overline{L}$. Conversely, L° is the set of axis numbers definitely *not* in the interpretation of L'_i due to the occurrence of $[\circ/a]$. $R^\bullet$ and R° are similar sets for $\overline{R}$ and R'_i. Finally, $B^\bullet$ and B° are the sets of axis numbers sure to be present and absent respectively in *both* L_i' and R_i' due to $\bullet$ and $\circ$ pairs in $\overline{B}$. Adim unification ensures that $B^\bullet$ and B° are well-defined. We can now write the interpretations of L_i' and R_i' explicitly as

$$\hat{L} = \{A \mid A = L^\bullet \cup B^\bullet \cup A' \text{ where } A' \subseteq (\mathrm{N} - L^\circ - B^\circ)\} \quad \text{and}$$
$$\hat{R} = \{A \mid A = R^\bullet \cup B^\bullet \cup A' \text{ where } A' \subseteq (\mathrm{N} - R^\circ - B^\circ)\}$$

i.e., the set of all sets of natural numbers that include at least the axes that must be present (due to $\bullet$-s in the types being unified) and exclude the ones that *must* be absent (due to $\circ$-s in the types being unified). Referring to the augmented S argument, we see that for any S', the interpretations of $U_{last}\, L_i$ and $U_{last}\, R_i$ are necessarily both the same subset of

$$\hat{L} \cap \hat{R} = \{I \mid I = R^\bullet \cup L^\bullet \cup B^\bullet \cup I' \text{ where } I' \subseteq (\mathrm{N} - L^\circ - R^\circ - B^\circ)\} \tag{4.7}$$

Otherwise, U_{last} could not unify E_{last}. Then observe that the substitutions for m and n in the processing step yield a permutation of

$$\overline{BLR}p$$

for both sides of the transformed e. The interpretation of this type is Equation 4.7 as desired, with one caveat. If a later substitution for p adds some pair with an axis in L, R, or B, then the corresponding adims must be unified; otherwise these types are invalid. Fortunately, such substitutions merely introduce new redundant pairs. The desired interpretation is assured by putting the transformed e back in Q, where it may become active once more due to substitution for p. Thus part (a) of the augmented S argument is proven. Similar reasoning yields

$$U\,[\overline{R}p/m]\,[\overline{L}p/n] \models Q_{last}$$

from which part (b) follows.

Regarding progress, it is easy to show that e must always have been active prior to this processing step and that the equation placed in Q by the step is not active.

Thus the occurrences of m and n have been removed from the count. Moreover, new occurrences of p in always-active equations each correspond to a disappearance of m or n, so they do not affect the count.

Two extensible maps, one map variable. If both L_i and R_i have the same map variable m (and no other) and at least one pair in L_i is different from every pair in R_i or *vice versa*, then form sets L, R, and B as above. Also, let p be a fresh map variable and $\overline{L}$ and $\overline{R}$ as above. Then replace S with

$$S\,[\overline{L}\overline{R}p/m]$$

and make associated replacements for m in Q. Make the same replacement in e, but note this duplicates the pairs in $\overline{L}$ on the left and $\overline{R}$ on the right. Thus, delete one of each duplication, and put the result back in Q.

The invariant is maintained in this step by a slight modification of the argument above.

To see progress, note that the equation replaced in Q is inactive after the step. Since e was always-active before the step, an occurrence of m in an always-active equation is lost, and the count decreases.

One extensible map. If either L_i or R_i has a single map variable m and the other has none, then assume without loss of generality that L_i has m (otherwise we can swap L_i and R_i). Form sets L, R, and B as above, and, unify adim pairs for axes in B just as in the case with no extensible map. Say $[D_1/a_1]\ldots[D_m/a_m]$ are the pairs associated with the axes in L from L_i and $[D'_1/a'_1]\ldots[D'_n/\mathrm{a'}_n]$ are the pairs associated with the axes in R from R_i. Then unify each of $D_1,\ldots D_n$ with $\circ$ and replace S with

$$S\,[\,[D'_1/a'_1]\ldots[D'_n/a'_n]/m]$$

and make associated replacements for m in Q.

The invariant is maintained by a conjunction of the arguments for ground maps and two extensible map types. We shall not repeat them. Moreover, a transformed e need not be placed in Q, because it contains no map variables that might be substituted with redundant pairs in the future.

Progress occurs because e was active and its occurrence of m is now gone from Q.

This completes the cases. There are two forms of equations not covered and that therefore might be left inactive when the algorithm terminates:

(a) Both sides contain the same non-redundant pairs and the same, single map variable.

(b) At least one side has more than one map variable.

Recall that for the algorithm to return a most general unifier as desired, we need $U \models Q$ for all substitutions U, where Q is the remaining set of inactive equations. This is certainly true of type (a) equations, but not for type (b). Fortunately, we can show that type (b) equations never remain in the particular sets we will generate.

4.5 Generating Axis Type Equations

The first phase of axis typing constructs a set of type equations that has a unifier if and only if program P has a type. A principal type of P itself and all its subterms are computed as a side-effect.

The algorithm manipulates yet one more kind of syntactic object—*typings*. A typing is a triple: $(\Gamma,\ E,\ T)$ where Γ is a type environment, E is a **svexp** subterm, and T is an axis type. It represents an assertion that E has some type that is a substitution instance of T in Γ, if it has any type at all. Figure 4.3 is the algorithm to construct equations for a single **svexp** function. It maintains a set Y of typings initialized with a single one that asserts the type of the entire function. In the

Algorithm: Axis type equation generation.

Inputs: A **svexp** function definition $D = (f_k\, x_1 \ldots x_n \equiv E)$.

Outputs: A set Q_0 of axis type equations.

Method: Set $Q_0 = \emptyset$;
Set $Y = \{(\Gamma, E, \langle\!\langle \emptyset, n, b\rangle\!\rangle)\}$
where $\Gamma = [\mathit{generic}(F_1)/f_1] \cdots [\mathit{generic}(F_{k-1})/f_{k-1}]$
$[\langle\!\langle \emptyset, n_1, b_1\rangle\!\rangle/x_1] \cdots [\langle\!\langle \emptyset, n_n, b_n\rangle\!\rangle/x_n]$
$[\langle\!\langle n_1, b_1\rangle\!\rangle, \ldots, \langle\!\langle n_n, b_n\rangle\!\rangle \rightarrow \langle\!\langle n, b\rangle\!\rangle/f]$;
While $Y \neq \emptyset$
{*Invariant.*}
Remove typing a $(\Gamma,\ E,\ T)$ from Y;
Execute the unique rule corresponding to the form of E
End while;
Return Q_0.

Figure 4.3: Axis type equation generation algorithm.

example of Figure 4.1, this is like writing the root of the tree at the bottom of the page. The algorithm then removes a typing from Y and uses the form of its subterm part to pick an *action rule*, which implements the typing rule for the subterm. This has two effects; it adds an equation to Q_0, and it adds zero or more typings to Y. The equations are the "notes" we referred to in Section 4.4 regarding $\equiv$ relationships that must hold to derive a type. Typings in Y represent the current fringe of the derivation tree. Eventually, Y consists entirely of typings that select the action rules for type axioms. The axiom rules add no new typings to Y, so the algorithm soon terminates. The typing rules are given in Figure 4.4.

4.5.1 Syntactic soundness and completeness

The goals of our careful development of axis type inference, namely soundness and completeness, can now be written as conditions on the equation set output of the algorithm:

$$
\begin{array}{rcl}
(\Gamma,\ \mathrm{Typ}\,c\,a\,E,\ T) & \Leftarrow & T \stackrel{?}{=} \langle\!\langle [\bullet/a]m,\ [\circ/c]n, b\rangle\!\rangle \quad (\Gamma, E, \langle\!\langle [d/a]m,\ [\bullet/c]n, b\rangle\!\rangle) \\[2ex]
(\Gamma,\ \mathrm{Proj}\,a\,c\,E,\ T) & \Leftarrow & T \stackrel{?}{=} \langle\!\langle [\circ/a]m,\ [\bullet/c]n, b\rangle\!\rangle \quad (\Gamma, E, \langle\!\langle [\bullet/a]m,\ [\circ/c]n, b\rangle\!\rangle) \\[2ex]
(\Gamma,\ \mathrm{Extr}_0\,c\,E,\ T) & \Leftarrow & T \stackrel{?}{=} \langle\!\langle m,\ [\circ/c]n, b\rangle\!\rangle \quad (\Gamma, E, \langle\!\langle m,\ [\bullet/c]n, b\rangle\!\rangle) \\[2ex]
(\Gamma,\ \mathrm{Extr}_1\,c\,E,\ T) & \Leftarrow & T \stackrel{?}{=} \langle\!\langle m,\ [\bullet/c]n, b\rangle\!\rangle \quad (\Gamma, E, \langle\!\langle m,\ [\bullet/c]n, b\rangle\!\rangle) \\[2ex]
(\Gamma,\ \mathrm{Red} +\ c\,E,\ T) & \Leftarrow & T \stackrel{?}{=} \langle\!\langle m, \emptyset, \mathbf{Int}\rangle\!\rangle \quad (\Gamma, E, \langle\!\langle m,\ [\bullet/c]\emptyset, \mathbf{Int}\rangle\!\rangle) \\[2ex]
(\Gamma,\ +\,E_1\,E_2,\ T) & \Leftarrow & T \stackrel{?}{=} \langle\!\langle m_1\,m_2, \emptyset, \mathbf{Int}\rangle\!\rangle \\
& & (\Gamma, E_1, \langle\!\langle m_1, \emptyset, \mathbf{Int}\rangle\!\rangle) \quad (\Gamma, E_2, \langle\!\langle m_2, \emptyset, \mathbf{Int}\rangle\!\rangle) \\[2ex]
(\Gamma,\ \mathrm{Join}\,c\,E_1\,E_2,\ T) & \Leftarrow & T \stackrel{?}{=} \langle\!\langle m_1\,m_2,\ [\bullet/c]n, b\rangle\!\rangle \\
& & (\Gamma, E_1, \langle\!\langle m_1,\ [d_1/c]n, b\rangle\!\rangle) \quad (\Gamma, E_2, \langle\!\langle m_2,\ [d_2/c]n, b\rangle\!\rangle) \\[2ex]
(\Gamma,\ \mathrm{Cond}\,B\,E_T\,E_F,\ T) & \Leftarrow & T \stackrel{?}{=} \langle\!\langle m_B\,m_T\,m_F, n, b\rangle\!\rangle \\
& & (\Gamma, B, \langle\!\langle m_B, \emptyset, \mathbf{Bool}\rangle\!\rangle) \\
& & (\Gamma, E_T, \langle\!\langle m_T, n, b\rangle\!\rangle) \quad (\Gamma, E_F, \langle\!\langle m_F, n, b\rangle\!\rangle) \\[2ex]
(\Gamma,\ x,\ T) & \Leftarrow & T \stackrel{?}{=} (\Gamma\,x) \\[2ex]
\left(\Gamma, \begin{bmatrix} x_{0,0} & \cdots & x_{0,q-1} \\ \vdots & \ddots & \vdots \\ x_{p-1,0} & \cdots & x_{p-1,q-1} \end{bmatrix}, T\right) & \Leftarrow & T \stackrel{?}{=} \langle\!\langle \emptyset,\ [\bullet/0]\,[\bullet/1], \mathbf{Int}\rangle\!\rangle \\
(\Gamma,\ f E_1 \cdots E_n,\ T) & \Leftarrow & T \stackrel{?}{=} \langle\!\langle m_1 \cdots m_n, n, b\rangle\!\rangle \\
& & (\Gamma, E_1, \langle\!\langle m_1, n_1, b_1\rangle\!\rangle) \cdots (\Gamma, E_n, \langle\!\langle m_n, n_n, b_n\rangle\!\rangle) \\
& & (\Gamma, f, \langle\!\langle n_1, b_1\rangle\!\rangle, \ldots, \langle\!\langle n_n, b_n\rangle\!\rangle \to \langle\!\langle n, b\rangle\!\rangle) \\[2ex]
(\Gamma,\ f,\ T) & \Leftarrow & \text{if } (\Gamma\,f) = \mathit{generic}(F) \text{ then} \\
& & \quad T \stackrel{?}{=} F' \quad \text{where } F' \text{ is } F \text{ with all variables} \\
& & \quad\phantom{T \stackrel{?}{=} F' \quad} \text{replaced by fresh ones} \\
& & \text{else} \\
& & \quad T \stackrel{?}{=} (\Gamma\,f)
\end{array}
$$

Figure 4.4: Function body action rules for axis typing.

Theorem 4.9 *When the algorithm of Figure 4.3 terminates with inputs as given, the following hold:*

Soundness. For any substitution U, if $U \models Q_0$, then

$$\emptyset \vdash f\colon (U\langle\!\langle m_1, b_1\rangle\!\rangle \ldots \langle\!\langle m_n, b_n\rangle\!\rangle \rightarrow \langle\!\langle m, b\rangle\!\rangle)$$

Completeness. For any axis type T such that $\emptyset \vdash f\colon T$, there exists a substitution U such that $U \models Q_0$ and $(U\langle\!\langle m_1, b_1\rangle\!\rangle \ldots \langle\!\langle m_n, b_n\rangle\!\rangle \rightarrow \langle\!\langle m, b\rangle\!\rangle) \equiv T$.

That is, every unifier of Q_0 yields a type that f actually has, and every type that f has may be obtained from some unifier for Q_0. These are sufficient because, as already shown, pass 2 will compute a single unifier more general than all others, or fail if no unifier exists.

Proof. The approach is straightforward and similar to the previous section's analysis of unification—we weaken the conditions above to achieve a loop invariant that may be proven true both initially and after each iteration. Intuitively, the set of typings represents work remaining do be done, and so is used in the invariant to account for equations not yet generated. Hence we say of substitution U that $U \models Y$ iff for each typing $(\Gamma, E, T) \in Y$, we have $(U \circ \Gamma) \vdash E : (U\,T)$. I.e. U is a witness that each typing represents at least one axis type derivation for its subterm part. Then we have:

Invariant. The following both hold:

Soundness. For any substitution U, if $U \models Q_0$ and $U \models Y$ then

$$\emptyset \vdash f\colon (U\langle\!\langle m_1, b_1\rangle\!\rangle \ldots \langle\!\langle m_n, b_n\rangle\!\rangle \rightarrow \langle\!\langle m, b\rangle\!\rangle).$$

Completeness. For any axis type T such that $\emptyset \vdash f\colon T$, there exists a substitution U such that $U \models Q_0$ and $U \models Y$ and $(U\langle\!\langle m_1, b_1\rangle\!\rangle \ldots \langle\!\langle m_n, b_n\rangle\!\rangle \rightarrow \langle\!\langle m, b\rangle\!\rangle) \equiv T$.

Progress criterion. Each action rule adds to Y only typings with smaller subterm parts than the typing that causes it to be selected.

The progress criterion, easily verified by checking the action rules, ensures that the typing set Y becomes empty after finitely many iterations. At this point the invariant is a proof of Theorem 4.9 as desired.

The invariant holds trivially on entry of the algorithm loop. It is preserved by iterations as shown in the next section. This completes the proof. □

4.5.2 Soundness and completeness cases

There is one case for each action rule in Figure 4.4. Assume for each that the invariant was true in the last iteration, where Y_{last} and Q_{0last} were the typings and equation set respectively and that the current iteration executes the action rule in boldface, yielding Y and Q_0.

Typ. To show soundness still holds, let U be a substitution such that $U \models Q_0$ and $U \models Y$. We wish to show $U \models Q_{0last}$ and $U \models Y_{last}$, for the desired result then follows by hypothesis. $U \models Q_{0last}$ because Q_0 includes all the equations of Q_{0last}. It remains to show $U \models Y_{last}$. The only typing in Y_{last} and not in Y is that on the left side of the rule. Thus we must prove $(U \circ \Gamma) \vdash \mathrm{Typ}\, c\, a\, E : (U\, T)$. For this, we have

$$(U \circ \Gamma) \vdash E : (U \langle\!\langle\, [d/a]m,\ [\bullet/c]n, b \rangle\!\rangle) \tag{4.8}$$

due to the typing just added to Y and our hypothesis $U \models Y$. With (4.8) as antecedent, typing rule *TYP* yields

$$(U \circ \Gamma) \vdash \mathrm{Typ}\, c\, a\, E : (U \langle\!\langle\, [\bullet/a]m,\ [\circ/c]n, b \rangle\!\rangle) \tag{4.9}$$

Finally, observe

$$(U\, T) \equiv (U \langle\!\langle\, [\bullet/a]m,\ [\circ/c]n, b \rangle\!\rangle)$$

due to the equation just added to Q_0, which with (4.9) yields the desired result.

Completeness is also preserved as follows. The inductive assumption implies that for any T such that $\emptyset \vdash f : T$, there must be U_{last} such that $U_{last} \models Q_{0last}$ and $U_{last} \models Y_{last}$. We must exhibit a U such that such that $U \models Q_0$ and $U \models Y$ and

$$(U \langle\!\langle m_1, b_1 \rangle\!\rangle \ldots \langle\!\langle m_n, b_n \rangle\!\rangle \rightarrow \langle\!\langle m, b \rangle\!\rangle) \equiv T$$

For this, note $(\Gamma, \mathrm{Typ}\, c\, a\, E, T) \in Y_{last}$, so $(U_{last}\, \Gamma) \vdash \mathrm{Typ}\, c\, a\, E : (U_{last}\, T)$. This can only follow from typing with rule *TYP* involving a substitution S such that $(S \langle\!\langle [d/a]m, [\bullet/c]n, b \rangle\!\rangle) \equiv (U_{last}\, T)$ and also $(U_{last}\, \Gamma) \vdash E : (S \langle\!\langle [d/a]m, [\bullet/c]n, b \rangle\!\rangle)$. Let $U = S \circ U_{last}$; then it is easy to show U has the desired properties. □

Proj, Extr, Red+, +, Join, Cond, and User Function Application. Proofs are similar to Typ using the respective typing rules.

Bound variables and constants. For soundness, we again must show $U \models Q_{0last}$ and $U \models Y_{last}$. The former holds because Q_0 includes all the equations of Q_{0last}, and $U \models Q_0$ by hypothesis. It remains to show $(U \circ \Gamma) \vdash x : (U\, T)$, because the only typing in Y_{last} that is not in Y is (Γ, x, T). We know $(U\, T) \equiv U(\Gamma\, x)$ by the new equation in Q_0 and the assumption $U \models Q_0$. Then from typing rule *VAR* we have

$$U \circ \Gamma \vdash x : (U \circ \Gamma)x \;\Rightarrow\; U \circ \Gamma \vdash x : U(\Gamma\, x) \;\Rightarrow\; U \circ \Gamma \vdash x : (U\, T) \qquad (4.10)$$

For completeness, we assume the existence of U_{last} and must exhibit U as for Typ above. Observe that Q_{0last} differs from Q_0 only in the added equation $T \stackrel{?}{=} (\Gamma\, x)$. Also, $(\Gamma,\, x,\, T) \in Y_{last}$, so $(U_{last}\, \Gamma) \vdash x : (U_{last}\, T)$. Since x can only be typed by applying rule *VAR*, there exists substitution S such that

$$S\,(U_{last}\, T) \equiv S\,(U_{last}\, \Gamma)x$$

It suffices to let $U = S \circ U_{last}$.

Constants are similar.

Function variables. If $(\Gamma\, f)$ has no *generic* tag, then the proof is as for bound variables. Arguments for the case $(\Gamma\, f) = generic(F)$ follow.

Soundness is the same as for bound variables, with the reasoning in (4.10) replaced by the following, which uses typing rules *VAR* and *INST* and the definition of *generic* on page 76.

$$\begin{aligned} U \circ \Gamma \vdash f : (U \circ \Gamma) f &\Rightarrow U \circ \Gamma \vdash f : (U\; generic(F)) \\ &\Rightarrow U \circ \Gamma \vdash f : generic(F) \\ &\Rightarrow U \circ \Gamma \vdash f : F' \end{aligned}$$

where F' is any substitution instance of F. We choose F' to be F with all variables replaced by fresh ones. In particular, these variables are not in the domain of U. Hence

$$U \circ \Gamma \vdash f : (U\, F')$$

We have $(U\, F') = (U\, T)$ from the new equation in Q_0 and the hypothesis, so

$$U \circ \Gamma \vdash f : (U\, T)$$

and $U \models Y_{last}$.

For completeness, we assume the existence of U_{last} and must again exhibit U. Observe that Q_{0last} differs from Q_0 only in the added equation $T \stackrel{?}{=} (\Gamma\, f)$. Note also that $(\Gamma,\, f,\, T) \in Y_{last}$, so $(U_{last}\, \Gamma) \vdash f : (U_{last}\, T)$. Since f can by typed only by rules *VAR* and *INST*, there exists substitution S such that

$$S\,(U_{last}\, T) \equiv S\, F'$$

where F' is F with all variables replaced by fresh ones. Let $U = S \circ U_{last}$.

4.6 Dimension Inference

The second type inference pass refines axis types to a new form that fully represents valid extents. Axis types already encode axis sets and the names of base domains. It remains to encode dimension functions. Consider these in the set-theoretic sense—as finite sets of pairs, one pair per axis set element with each pair containing an axis number and a size. There is an immediate correspondence between these pairs and axis type pairs containing $\bullet$. A pair with $\bullet$ encodes a mapping from the axis number it contains to some size in $\mathbf{N}$, though we do not yet know what that size is. The job of the second pass is thus to replace $\bullet$ with a representation of size.

4.6.1 Dimensions

We require a language to encode sizes. Since a single **svexp** function will often be applied to different-sized operands, numerals alone will not do. We need to encode arithmetic constraints on dimension function range values, much as maps with adim and map variables encode constraints on axis sets. This leads us to the simple arithmetic expressions with which the closure theorems 3.8 and 3.12 are written. The following syntax is adequately expressive:

$$\begin{aligned} \langle \mathit{dim} \rangle \ ::= \ & 0 \mid 1 \mid 2 \ldots \\ & \mid \langle \mathit{dim} \rangle + \langle \mathit{dim} \rangle \mid \langle \mathit{dim} \rangle \dot{-} \langle \mathit{dim} \rangle \mid \ \min \{ \langle \mathit{dim} \rangle^{+} \} \\ & \mid \langle \mathit{dim\ variable} \rangle \mid \langle \mathit{dependent\text{-}dim} \rangle (\ \langle \mathit{dim} \rangle^{*} \ ) \mid \circ \end{aligned} \tag{4.11}$$

Thus $\langle \mathit{dim} \rangle$ includes numerals and arithmetic with the conventional interpretation over $\mathbf{N}$. In particular, $a \dot{-} b$ denotes 0 if $a < b$ and $a - b$ otherwise as in Theorem 3.8. Dimension variables are written d or d_k or d_{jk} by convention. A *dependent dimension* denotes application of an unspecified function $D_k : \mathbf{N} \times \cdots \times \mathbf{N} \rightarrow \mathbf{N}$ to dimensions. Finally, there is the familiar $\circ$, which denotes absent axes as for axis types. Dimension variables *cannot* take this value.

Replacing $\langle \mathit{adim} \rangle$ in Figure 4.1 with $\langle \mathit{dim} \rangle$ yields full-fledged *types*. The interpretation of types specializes the axis type interpretation given in Section 4.2. A ground map M now denotes an axis set–dimension function pair. The axis set is

$$A = \{ a \mid \ [D/a] \text{ occurs in } M \text{ and } D \neq \circ \}.$$

The dimension function α is given for all $a \in A$ by

$$(\alpha\, a) = D \text{ if } [D/a] \text{ occurs in } M$$

The characterization of $\equiv$ for full types includes equality of dimensions in the obvious sense; e.g.

$$[1+2/a]\,[\circ/a_2]\emptyset \equiv\; [\circ/a_1]\,[3/a]\emptyset$$

A dimension is a *ground dimension* if it contains no dimension variables or dependent dimensions. Interpretation of ground and non-ground types is then as for axis types.

4.6.2 Inference rules

The reader can easily verify that soundness of the type inference rules for **svexp** primitives in Table 4.5 follow from Theorems 3.8 and 3.12 and the semantic equations as the axis typing rules did, except soundness is now unqualified. That is, when operands to Join and Cond are proper antecedents to one of the rules *dJOINx* or of *dCOND*, then the result cannot be $typerr_2$ (or $typerr_1$).

Table 4.5 includes two rules for Typ and four for Join. These treat separately the "diagonal" case of Typ (Example 3.5) and the possible promotions of Join arguments (Section 2.3.5). At first glance, this presents a problem for the second inference pass, which builds proofs just as the first pass (see Figure 4.1). On reaching a Typ or Join, with which rule will the proof succeed? The inference algorithm consults the axis types of arguments for the answer.

Constants and variable typing rules are given in Table 4.6. Rule *CONST* for constant svals of any base type $\mathbf{B}_k$ is straightforward, but the separate rules for bound and function variables merit discussion. Indeed, we could use *BVAR* for all variables and achieve soundness, but many useful programs would then be untypable.

Example 4.10 Consider the following code to reverse components along axis 0 of the input.

Table 4.5: Primitive typing rules.

$$\frac{\Gamma \vdash E : \langle\!\langle\, [\circ/a]m,\ \ [d/c]n,\ b \rangle\!\rangle}{\Gamma \vdash \mathrm{Typ}\, c\, a\, E : \langle\!\langle\, [d/a]m,\ \ [\circ/c]n,\ b \rangle\!\rangle} \qquad dTYPa$$

$$\frac{\Gamma \vdash E : \langle\!\langle\, [d_1/a]m,\ \ [d_2/c]n,\ b \rangle\!\rangle}{\Gamma \vdash \mathrm{Typ}\, c\, a\, E : \langle\!\langle\, [\min\,\{d_1, d_2\}/a]m,\ \ [\circ/c]n,\ b \rangle\!\rangle} \qquad dTYPb$$

$$\frac{\Gamma \vdash E : \langle\!\langle\, [d/a]m,\ \ [\circ/c]n,\ b \rangle\!\rangle}{\Gamma \vdash \mathrm{Proj}\ a\ c\, E : \langle\!\langle\, [\circ/a]m,\ \ [d/c]n,\ b \rangle\!\rangle} \qquad dPROJ$$

$$\frac{\Gamma \vdash E : \langle\!\langle m,\ \ [d/c]n,\ b \rangle\!\rangle}{\Gamma \vdash \mathrm{Extr}_0\ c\, E : \langle\!\langle m,\ \ [\circ/c]n,\ b \rangle\!\rangle} \qquad dEXTR0$$

$$\frac{\Gamma \vdash E : \langle\!\langle m,\ \ [d/c]n,\ b \rangle\!\rangle}{\Gamma \vdash \mathrm{Extr}_1\ c\, E : \langle\!\langle m,\ \ [d \dot{-} 1/c]n,\ b \rangle\!\rangle} \qquad dEXTR1$$

$$\frac{\Gamma \vdash E : \langle\!\langle m,\ \ [d/c]\emptyset,\ \mathbf{Int} \rangle\!\rangle}{\Gamma \vdash \mathrm{Red} +\ \ c\, E : \langle\!\langle m,\ \emptyset,\ \mathbf{Int} \rangle\!\rangle} \qquad dRED+$$

$$\frac{\Gamma \vdash E_1 : \langle\!\langle m_1,\ \emptyset,\ \mathbf{Int} \rangle\!\rangle,\ E_2 : \langle\!\langle m_2,\ \emptyset,\ \mathbf{Int} \rangle\!\rangle}{\Gamma \vdash\ +\ E_1\ E_2 : \langle\!\langle m_1\, m_2,\ \emptyset,\ \mathbf{Int} \rangle\!\rangle} \qquad dBIN+$$

$$\frac{\Gamma \vdash E_1 : \langle\!\langle m_1,\ \ [d_1/c]n,\ b \rangle\!\rangle,\ E_2 : \langle\!\langle m_2,\ \ [d_2/c]n,\ b \rangle\!\rangle}{\Gamma \vdash \mathrm{Join}\ c\, E_1\ E_2 : \langle\!\langle m_1\, m_2,\ \ [d_1{+}d_2/c]n,\ b \rangle\!\rangle} \qquad dJOINa$$

$$\frac{\Gamma \vdash E_1 : \langle\!\langle m_1,\ \ [d_1/c]n,\ b \rangle\!\rangle,\ E_2 : \langle\!\langle m_2,\ \ [\circ/c]n,\ b \rangle\!\rangle}{\Gamma \vdash \mathrm{Join}\ c\, E_1\ E_2 : \langle\!\langle m_1\, m_2,\ \ [d_1{+}1/c]n,\ b \rangle\!\rangle} \qquad dJOINb$$

$$\frac{\Gamma \vdash E_1 : \langle\!\langle m_1,\ \ [\circ/c]n,\ b \rangle\!\rangle,\ E_2 : \langle\!\langle m_2,\ \ [d_2/c]n,\ b \rangle\!\rangle}{\Gamma \vdash \mathrm{Join}\ c\, E_1\ E_2 : \langle\!\langle m_1\, m_2,\ \ [1{+}d_2/c]n,\ b \rangle\!\rangle} \qquad dJOINc$$

$$\frac{\Gamma \vdash E_1 : \langle\!\langle m_1,\ \ [\circ/c]n,\ b \rangle\!\rangle,\ E_2 : \langle\!\langle m_2,\ \ [\circ/c]n,\ b \rangle\!\rangle}{\Gamma \vdash \mathrm{Join}\ c\, E_1\ E_2 : \langle\!\langle m_1\, m_2,\ \ [2/c]n,\ b \rangle\!\rangle} \qquad dJOINd$$

$$\begin{aligned} \mathit{rev}\ x \equiv\ & \text{if (NumOf } x \text{ 0 1) then } x \\ & \text{else Join 0 } (\mathit{rev}\ \text{Extr}_1^0\ 0\ x)\ (\text{Extr}_0^0\ 0\ x) \end{aligned}$$

We "guess" the assumption that *rev* acts on d-vectors, i.e.

$$\Gamma = [\langle\!\langle [d/0]\emptyset, b\rangle\!\rangle / x]\ \ [\langle\!\langle [d/0]\emptyset, b\rangle\!\rangle \rightarrow \langle\!\langle [d/0]\emptyset, b\rangle\!\rangle / \mathit{rev}]$$

so that we can prove *rev* takes d-vectors to d-vectors, i.e.

$$\mathit{rev} : \langle\!\langle [d/0]\emptyset, b\rangle\!\rangle \rightarrow \langle\!\langle [d/0]\emptyset, b\rangle\!\rangle \quad \text{as defined above.}$$

But the proof succeeds only if we can establish, as an intermediate step, that the recursive binding of *rev* has type

$$\langle\!\langle [d \dot{-} 1/0]\emptyset, b\rangle\!\rangle \rightarrow \langle\!\langle [d \dot{-} 1/0]\emptyset, b\rangle\!\rangle$$

□

Thus we must be free to instantiate dimensions of the types of even non-*generic* function variables. Let S be any substitution whose domain includes no free map or base type variables of T. Then $(S\,T)$ is a *dimension substitution instance* of T. Rule *FVAR* allows such instantiation of function types. The remainder of typing rules are as for axis types.

Theorem 4.11 (Semantic soundness of typing) *Let P be an* **svexp** *program. Then $\emptyset \vdash P : T$ implies $\mathcal{P}[\![P]\!] : T$.*

Proof. Almost identical to that for Theorem 4.5. The only significant difference is that typerr_2 is no longer a possible value for a term with a type because the rules *dJOINx* and *dCOND* are sound without qualification. □

4.6.3 Type unification

Axis type unification (Figure 4.2) needs only a small modification to unify general types. In addition to the substitution S, the modification constructs a set Z of

Table 4.6: Constant and variable typing rules.

$$\Gamma \vdash \begin{bmatrix} x_{0,0} & \cdots & x_{0,q-1} \\ \vdots & \ddots & \vdots \\ x_{p,0} & \cdots & x_{p-1,q-1} \end{bmatrix} : \langle\!\langle \emptyset,\ [p/0]\,[q/1]\emptyset, \mathbf{B}_k \rangle\!\rangle \qquad \textit{dCONST}$$

$$\Gamma \vdash x : (\Gamma\, x) \quad \text{(bound variable expression)} \qquad \textit{dBVAR}$$

$$\Gamma \vdash f : F' \quad \text{where } F' \text{ is a dimension substitution instance of } (\Gamma\, f) \qquad \textit{dFVAR}$$

dimension equality constraints, which are just assertions of equality over $\mathbf{N}$, written $D_1 \stackrel{!}{=} D_2$. We replace the adim unification "subroutine" on page 86 with the following:

Dimension unification. Unify dimensions D and D' as follows: If $D = D' = \circ$, we are done. If either D or D' is $\circ$ and the other is not, return *fail* for the top level unification of Q_0. Otherwise add $D \stackrel{!}{=} D'$ to Z.

What of soundness and completeness of the modified algorithm? We are concerned only about a weak form of these. The algorithm is sound in the sense that if T is any substitution for dimension variables such that $T \models Z$, then $T \circ S \models Q_0$. It is complete in the sense that, for any S' binding only map variables and T binding only dimension variables, if $S' \circ T \models Q_0$, then there exists another substitition U such that $S \circ U \equiv S'$. Intuitively, we are merely assuming the existence of solutions for Z. This is reasonable because the assertions in Z can be verified by "checking code" in the compiled program.[5] Of course, we could say the same about Q_0; the whole point of type inference is to eliminate checking code. Alas, though much can

[5]The exception is dependent dimensions, which are discussed below.

be done to reduce Z so that checking code is minimal, some checks will be required.[6] We can easily verify the weak forms of soundness and completeness by proving the following for each iteration as in Section 4.4.

Invariant. Let T be a substitution with domain restricted to the free (dimension) variables of Z such that $T \models Z$.

Soundness. For any substitution S' such that $S' \models Q_0$, we have $S \succeq S'$. In particular, we can exhibit a unifier U of Q such that $U \circ S \circ T \equiv S'$.

Completeness. For any substitution U, if $U \models Q$, then $U \circ S \circ T \models Q_0$.

4.6.4 Most general function types

We next consider how to implement the liberal dimension instantiation allowed by rule *FVAR* for a function variable reference f. In the first type inference pass discussed in Sections 4.2 through 4.5, we instantiate generic function axis types by replacing variables with fresh ones. This is still fine for generic types in the second pass, those at non-recursive function calls. We merely replace dimension variables as well as map and base type variables.

Things are more difficult if f is a recursive binding. In the first pass, we merely assume (i.e. add to Γ)

$$f : \langle\!\langle n_1, b_1 \rangle\!\rangle, \ldots, \langle\!\langle n_n, b_n \rangle\!\rangle \rightarrow \langle\!\langle n, b \rangle\!\rangle \tag{4.12}$$

by rule *ABS* in Table 4.4, where the important point is that the type is non-generic. Thus, when inference is complete, all arguments of recursive calls to f have axis types equivalent with respect to $\equiv$ to the corresponding formal parameter type of f.

[6] **svexp** as given actually requires no checking code. Because its only "data" are constants, all dimensions can be inferred exactly if we make dimension constraints part of function types and apply constraints symbolically to arguments at each function call. This is a desirable extension of the algorithm presented here. However, the addition of a simple matrix input primitive to **svexp** makes checking code necessary.

While this is fine for axis types, it is too strong for the current purpose. That is, if f's formal parameter is an inner column vector of length d, the axis typing rules for the first typing pass properly insist that arguments of all recursive calls to f are also column vectors, but it would be over-zealous for the second pass to insist that arguments of all recursive calls to f have length d. For instance, the recursive call in (4.12) would be a type error in this case.

A remedy analogous to the way substitution of fresh variables is used to instantiate generic types is to replace *only* dimension variables in the assumed type of f (the guess in Example 4.10) with fresh ones at each recursive call. However, this requires a prescient guess for the assumption. The type in (4.12), which made a fine assumption for the type of f in pass one is no help; it has no dimension variables to replace! Indeed, it seems we need to replace dimension variables in the very type we are trying to infer.

This nasty circularity is another motive for two pass type inference, for a principal *axis type* of f is sufficient to unravel it.

Definition 4.12 (Type assumption for f) *Let* $\langle\!\langle M_1, \tau_1\rangle\!\rangle, \ldots, \langle\!\langle M_n, \tau_n\rangle\!\rangle \to \langle\!\langle M, \tau\rangle\!\rangle$ *be a principal axis type of f. For each axis map M_k, let M_k^{dim} be a dimension map the same as M_k, except replace each pair $[\bullet/c]$ with $[d_{kc}/c]$. Let $\overline{D}$ be the dimension variables thus introduced arranged in a canonical order and let M^{dep} be M with each pair $[\bullet/c]$ replaced by a pair with a dependent dimension $[D_c(\overline{D})/c]$. Then the type*

$$\langle\!\langle M_1^{dim}, \tau_1\rangle\!\rangle, \ldots, \langle\!\langle M_n^{dim}, \tau_n\rangle\!\rangle \to \langle\!\langle M^{dep}, \tau\rangle\!\rangle$$

is a type assumption for f. □

Recall that D_c denotes an unspecified function over **N**. Hence, this type is at least as general as any type of f. Without affecting the power of the type system, we can

Algorithm: Type equation generation.

Inputs: A **svexp** function definition $D = (f_k\, x_1 \ldots x_n \equiv E)$.

Outputs: A set Q_0 of type equations.

Method: Set $Q_0 = \emptyset$;
Set $Y = \{(\Gamma, E, \langle\!\langle \emptyset, M^{dep}, b\rangle\!\rangle)\ \}$
where $\Gamma = [generic(F_1)/f_1] \cdots [generic(F_{k-1})/f_{k-1}]$
$[\langle\!\langle \emptyset, M_1^{dim}, \tau_1\rangle\!\rangle/x_1] \cdots [\langle\!\langle \emptyset, M_n^{dim}, \tau_n\rangle\!\rangle/x_n]$
$[\langle\!\langle M_1^{dim}, \tau_1\rangle\!\rangle, \ldots, \langle\!\langle M_n^{dim}, \tau_n\rangle\!\rangle \rightarrow \langle\!\langle M^{dep}, \tau\rangle\!\rangle/f]$;
(see Definition 4.12)
While $Y \neq \emptyset$
{*Invariant.*}
Remove typing a $(\Gamma,\ E,\ T)$ from Y;
Execute the unique rule corresponding to the form of E
End while;
{*Invariant.*}
Return Q_0.

Figure 4.5: Type equation generation algorithm.

replace rule *ABS* with

$$\frac{\begin{array}{l}\emptyset \vdash f_1 : F_1, \ldots\ \emptyset \vdash f_{k-1} : F_{k-1}, \\ [generic(F_1)/f_1]\ \ldots\ [generic(F_{k-1})/f_{k-1}] \\ \qquad [\langle\!\langle M_1^{dim}, \tau_1\rangle\!\rangle, \ldots, \langle\!\langle M_n^{dim}, \tau_n\rangle\!\rangle \rightarrow \langle\!\langle M^{dep}, \tau\rangle\!\rangle/f_k] \\ \qquad [\langle\!\langle \emptyset, M_1^{dim}, \tau_1\rangle\!\rangle/x_1]\ \ldots\ [\langle\!\langle \emptyset, M_n^{dim}, \tau_n\rangle\!\rangle/x_n]\ \vdash\ E : \langle\!\langle \emptyset, M^{dep}, \tau\rangle\!\rangle\end{array}}{\emptyset \vdash f_k : \langle\!\langle M_1^{dim}, \tau_1\rangle\!\rangle, \ldots, \langle\!\langle M_n^{dim}, \tau_n\rangle\!\rangle \rightarrow \langle\!\langle M^{dep}, \tau\rangle\!\rangle\ (f_k \text{ in } f_k\, x_1 \ldots, x_n \equiv E)}\ dABS$$

4.6.5 Generating Type Equations

The algorithm for generating type equations for a **svexp** function f_k is given in Figure 4.5. It is the same as Figure 4.3 for axis types, except the initial assumption of the function type reflects *dABS*. The corresponding action rules are in Figure 4.6 and Figure 4.7. These are derived from the typing rules of Tables 4.5 and 4.6. Note

$$(\Gamma,\ \mathrm{Typ}\, c\, a\, (E : \langle\!\langle M_A, M_C, \tau \rangle\!\rangle),\ T) \Leftarrow$$
$$\text{if } [\circ/a] \in M_A \text{ then}$$
$$T \stackrel{?}{=} \langle\!\langle\, [d/a]m,\ [\circ/c]n, b \rangle\!\rangle \quad (\Gamma, E, \langle\!\langle\, [\circ/a]m,\ [d/c]n, b \rangle\!\rangle)$$
$$\text{else}$$
$$T \stackrel{?}{=} \langle\!\langle\, [\min\{d_1, d_2\}/a]m,\ [\circ/c]n, b \rangle\!\rangle \quad (\Gamma, E, \langle\!\langle\, [d_1/a]m,\ [d_2/c]n, b \rangle\!\rangle)$$

$$(\Gamma,\ \mathrm{Proj}\, a\, c E,\ T) \ \Leftarrow\ T \stackrel{?}{=} \langle\!\langle\, [\circ/a]m,\ [d/c]n, b \rangle\!\rangle \quad (\Gamma, E, \langle\!\langle\, [d/a]m,\ [\circ/c]n, b \rangle\!\rangle)$$

$$(\Gamma,\ \mathrm{Extr}_0\, c E,\ T) \ \Leftarrow\ T \stackrel{?}{=} \langle\!\langle m,\ [\circ/c]n, b \rangle\!\rangle \quad (\Gamma, E, \langle\!\langle m,\ [d/c]n, b \rangle\!\rangle)$$

$$(\Gamma,\ \mathrm{Extr}_1\, c E,\ T) \ \Leftarrow\ T \stackrel{?}{=} \langle\!\langle m,\ [d \mathbin{\dot{-}} 1/c]n, b \rangle\!\rangle \quad (\Gamma, E, \langle\!\langle m,\ [d/c]n, b \rangle\!\rangle)$$

$$(\Gamma,\ \mathrm{Red}+\ c E,\ T) \ \Leftarrow\ T \stackrel{?}{=} \langle\!\langle m, \emptyset, \mathbf{Int} \rangle\!\rangle \quad (\Gamma, E, \langle\!\langle m,\ [d/c]\emptyset, \mathbf{Int} \rangle\!\rangle)$$

$$(\Gamma,\ + E_1\, E_2,\ T) \ \Leftarrow\ T \stackrel{?}{=} \langle\!\langle m_1\, m_2, \emptyset, \mathbf{Int} \rangle\!\rangle$$
$$(\Gamma, E_1, \langle\!\langle m_1, \emptyset, \mathbf{Int} \rangle\!\rangle) \quad (\Gamma, E_2, \langle\!\langle m_2, \emptyset, \mathbf{Int} \rangle\!\rangle)$$

$$(\Gamma,\ \mathrm{Join}\, c\, (E_1 : \langle\!\langle M_{A1}, M_{C1}, \tau_1 \rangle\!\rangle)\, (E_2 : \langle\!\langle M_{A2}, M_{C2}, \tau_2 \rangle\!\rangle),\ T) \Leftarrow$$
$$\text{if } [\bullet/c] \in M_{C1} \text{ and } [\bullet/c] \in M_{C2} \text{ then}$$
$$T \stackrel{?}{=} \langle\!\langle m_1\, m_2,\ [d_1 + d_2/c]n, b \rangle\!\rangle\ (\Gamma, E_1, \langle\!\langle m_1,\ [d_1/c]n, b \rangle\!\rangle)\ (\Gamma, E_2, \langle\!\langle m_2,\ [d_2/c]n, b \rangle\!\rangle)$$
$$\text{else if } [\bullet/c] \in M_{C1} \text{ and } [\circ/c] \in M_{C2} \text{ then}$$
$$T \stackrel{?}{=} \langle\!\langle m_1\, m_2,\ [d_1 + 1/c]n, b \rangle\!\rangle\ (\Gamma, E_1, \langle\!\langle m_1,\ [d_1/c]n, b \rangle\!\rangle)\ (\Gamma, E_2, \langle\!\langle m_2,\ [\circ/c]n, b \rangle\!\rangle)$$
$$\text{else if } [\circ/c] \in M_{C1} \text{ and } [\bullet/c] \in M_{C2} \text{ then}$$
$$T \stackrel{?}{=} \langle\!\langle m_1\, m_2,\ [1 + d_2/c]n, b \rangle\!\rangle\ (\Gamma, E_1, \langle\!\langle m_1,\ [\circ/c]n, b \rangle\!\rangle)\ (\Gamma, E_2, \langle\!\langle m_2,\ [d_2/c]n, b \rangle\!\rangle)$$
$$\text{else}$$
$$T \stackrel{?}{=} \langle\!\langle m_1\, m_2,\ [2/c]n, b \rangle\!\rangle\ (\Gamma, E_1, \langle\!\langle m_1,\ [\circ/c]n, b \rangle\!\rangle)\ (\Gamma, E_2, \langle\!\langle m_2,\ [\circ/c]n, b \rangle\!\rangle)$$

Figure 4.6: Function body action rules for typing (part 1).

$$
\begin{aligned}
(\Gamma,\ \mathrm{Cond}\, B\, E_T\, E_F,\ T) &\Leftarrow\ T \stackrel{?}{=} \langle\!\langle m_B\, m_T\, m_F, n, b \rangle\!\rangle \\
&\ \ (\Gamma, B, \langle\!\langle m_B, \emptyset, \mathbf{Bool} \rangle\!\rangle) \\
&\ \ (\Gamma, E_T, \langle\!\langle m_T, n, b \rangle\!\rangle)\quad (\Gamma, E_F, \langle\!\langle m_F, n, b \rangle\!\rangle) \\[1em]
(\Gamma,\ x,\ T) &\Leftarrow\ T \stackrel{?}{=} (\Gamma\, x) \\[1em]
\left(\Gamma,\ \begin{bmatrix} x_{0,0} & \cdots & x_{0,q-1} \\ \vdots & \ddots & \vdots \\ x_{p-1,0} & \cdots & x_{p-1,q-1} \end{bmatrix},\ T\right) &\Leftarrow\ T \stackrel{?}{=} \langle\!\langle \emptyset,\ [p/0]\,[q/1], \mathbf{Int} \rangle\!\rangle \\
(\Gamma,\ f E_1 \cdots E_n,\ T) &\Leftarrow\ T \stackrel{?}{=} \langle\!\langle m_1 \cdots m_n, n, b \rangle\!\rangle \\
&\ \ (\Gamma, E_1, \langle\!\langle m_1, n_1, b_1 \rangle\!\rangle) \cdots (\Gamma, E_n, \langle\!\langle m_n, n_n, b_n \rangle\!\rangle) \\
&\ \ (\Gamma, f, \langle\!\langle n_1, b_1 \rangle\!\rangle, \ldots, \langle\!\langle n_n, b_n \rangle\!\rangle \to \langle\!\langle n, b \rangle\!\rangle) \\[1em]
(\Gamma,\ f,\ T) &\Leftarrow\ \text{if } (\Gamma\, f) = \mathit{generic}(F) \text{ then} \\
&\ \ \quad T \stackrel{?}{=} F' \quad \text{where } F' \text{ is } F \text{ with all variables replaced by fresh ones} \\
&\ \ \text{else} \\
&\ \ \quad T \stackrel{?}{=} F' \quad \text{where } F' \text{ is } F \text{ with dimension variables replaced by fresh ones}
\end{aligned}
$$

Figure 4.7: Function body action rules for typing (part 2).

the rules for Typ and Join consult the axis types of arguments to select a case corresponding one of the associated typing rules for these constructs. The proof of soundness and completeness for this algorithm is almost identical to that for Theorem 4.9, and we omit it.

4.6.6 Simplifying dimension equality constraints

After type equations have been generated and solved by unification, we are left with the set Z of dimension equality constraints. It is not generally possible to solve these in the complete way unification solves type equations, where a substitution exactly encodes all solutions. However, we can simpify Z symbolically in several ways.

(a) *Apply identities over* N. In particular, fold constants and replace e.g. $(d_1 + d_2) \dot{-} d_2$ with d_1.

(b) *Eliminate variables.* If there exists an equation of the form $d \stackrel{!}{=} D$ in Z, wherein d does not occur in D, then replace d with D in all equations and types. Practically speaking, we want to eliminate all variables except those introduced in the type assumption (Definition 4.12 and Figure 4.5). Otherwise, we may prove e.g. that

$$rev : \langle\!\langle\, [d + 1/0], b \rangle\!\rangle \to \langle\!\langle\, [d + 1/0], b \rangle\!\rangle$$

rather than the more useful

$$rev : \langle\!\langle\, [d_{01}/0], b \rangle\!\rangle \to \langle\!\langle\, [d_{01}/0], b \rangle\!\rangle$$

(c) *Throw out tautologies.* Equations of the form $D_1 \stackrel{!}{=} D_2$ where $D_1 = D_2$ (e.g. $3 = 3$ and $d_1 + d_2 \stackrel{!}{=} d_2 + d_1$) are superfluous and may be removed from Z.

(d) *Report* fail *for contradictions.* If an equation such as $2 \stackrel{!}{=} 1$ arises, the function may return $typerr_2$ and must be discarded as erroneous.

(e) *Eliminate dependent dimensions.* In general, equations in Z will contain subexpressions of the form $D_c(\overline{E})$, where $\overline{E}$ is a sequence of $\langle dim \rangle$ expressions. Such terms are the dimensions of recursive function call results. They cannot be translated to checking code or evaluated symbolically unless the function that D_c denotes is known.

Item (e) provides the last task for the type checker. It must derive D_c for each axis c in the result type of f. The following theorem at once shows that this is always possible (if f terminates) and suggests how to do it.

Theorem 4.13 *Let f and its type assumption be as in Definition 4.12. Let Z be the dimension equality constraints from typing f. Further suppose variables not in $\overline{D}$*

have been eliminated wherever possible in Z. Suppose f terminates for at least one input (i.e. its semantic value is not $\perp$). Then for each D_c in M^{dep}, there exists in Z an equation of the form $D_c(\overline{D}) = E_c$, where E_c is a dimension expression with no dependent dimensions and whose variables are all in $\overline{D}$.

Equations $D_c(\overline{D}) = E_c$ are important because they are closed definitions for dependent dimension functions. We can use them as checking code or, more fruitfully, to eliminate dependent dimensions from Z altogether. That is, for all subexpressions $D_c(\overline{E})$ appearing in Z, we can substitute E_c wherein all variables in $\overline{D}$ have been replaced by the respective expressions in $\overline{E}$. It remains to prove Theorem 4.13.

Proof. A **svexp** term has the property *useful* if it satisfies the following inductive definition:

- Bound variable references and constants are useful.
- Non-recursive user function applications and primitive applications excepting Cond are useful if their arguments are useful.
- A Cond application is useful if its predicate expression is useful and at least one of its branches is useful.

In particular, code including recursive function calls is *not* useful, unless the call is embedded in a Cond branch, and the other branch is useful. It is straightforward to show using the full semantics of **svexp** that any function f whose semantic value is not $\perp$ has a useful body. Operationally, this is equivalent to saying that f is free of technical errors such as undefined variable references and that it terminates for some input.

A *typing* is useful if its **svexp** subterm part is useful.

For convenience, we assume that the unification algorithm is applied to each new equation as it is generated by Figure 4.5. Thus we can reason about the equations introduced in Z by application of each action rule.

We show that the following invariant is true for each iteration in Figure 4.5.

Invariant. One of the following holds regarding each dependent dimension function D_c:

(a) A pair $[D_c(\overline{D})/a]$ appears in a useful typing in Y.

(b) Z (with variables eliminated) contains an equation $D_c(\overline{D}) = E$, wherein E has no dependent dimensions and each variable d is either in $\overline{D}$ or appears in a pair $[d/a]$ in a useful typing.

Since no useful typings remain when the algorithm terminates successfully, this proves the theorem as desired.

Before the first iteration, Z is empty and Y contains only the useful typing

$$(\Gamma, E, \langle\!\langle \emptyset, M^{dep}, b \rangle\!\rangle)$$

which satisfies (a).

Iterations that act on non-useful typings do not affect the invariant.

Suppose the iteration selects a useful typing for Proj $a\,c$. Then the new typing introduced to Y is certain to be useful. Consider

$$T \stackrel{?}{=} \langle\!\langle [\circ/a]m,\ [d/c]n, b \rangle\!\rangle \quad \text{and} \quad (\Gamma, E, \langle\!\langle [d/a]m,\ [\circ/c]n, b \rangle\!\rangle)$$

the new equation and typing introduced by the action rule. Unification of variables m and n ensures all pairs in T except one for axis c also occur in the new typing after unification. Thus the invariant is preserved for all these pairs. Regarding the pair for axis c, suppose it is of the form $[D_c(\overline{D})/c]$, i.e. it supported invariant (a) in the last iteration. Call this an (a)-pair. Then unification introduces the equation $D_c(\overline{D}) \stackrel{!}{=} d$ into Z, but d occurs in a pair $[d/a]$ in the new, useful typing in Y, so the invariant is preserved. Similarly, suppose the pair for c in T is of the form $[d'/c]$, where d' appears in the right hand side of an equation $D_c(\overline{D}) \stackrel{!}{=} E$ in Z. I.e. this pair supported invariant (b) in the last iteration. Call it a (b)-pair. Then unification introduces the new equation $d = d'$. By variable substitution, we replace d' by d in E, and the invariant is again preserved.

Similar reasoning establishes the invariant for all primitives and non-recursive user function calls. For Cond, we need the additional fact, due to the definition of *useful*, that at least one of the branch typings introduced in Y is useful, and this typing preserves the invariant.

At a bound variable reference, each pair in T unifies with a pair whose dimension is a variable in $\overline{D}$. Thus if a pair in T is an (a)-pair as above, unification introduces an equation in Z, where $d \in \overline{D}$; thus the invariant stands. Similarly, if a pair in T is a (b)-pair, unification introduces an equation $d' = d$, where $d \in \overline{D}$. Variable elimination replaces d' with d in E, and the invariant holds again. Constants are similar, introducing equations in Z of the form $D_c(\overline{D}) \stackrel{!}{=} k$, k a numeral, for (a)-pairs and $d' = k$ for (b)-pairs. □

For Example 4.10, axis typing yields $rev : \langle\!\langle\, [\bullet/0]n, b \rangle\!\rangle \rightarrow \langle\!\langle\, [\bullet/0]n, b \rangle\!\rangle$. Hence the type assumption for rev is $\langle\!\langle\, [d_{10}/0]n, b \rangle\!\rangle \rightarrow \langle\!\langle\, [D_0(d_{10})/0]n, b \rangle\!\rangle$. After type checking and variable elimination, we have $Z = \{D_0(d_{10}) \stackrel{!}{=} d_{10},\ D_0(d_{10}) \stackrel{!}{=} 1 + D_0(d_{10} \dot{-} 1)\}$. Using the first equation twice, we obtain $Z = \{d_{10} \stackrel{!}{=} d_{10},\ d_{10} \stackrel{!}{=} 1 + (d_{10} \dot{-} 1)\}$. The first equation may be immediately eliminated as a tautology. The second is true whenever $d_{10} > 0$. We compile this as a run time check.

Chapter 5

Code Generation and Improvement

In this chapter, we set out to compile **svexp** programs to an intermediate form that translates easily to machine code for a variety of single and multiple processor architectures. As an example, we give details for translation to the C language [KR78], which serves well as an abstract assembler code.

Compilation of **svexp**, as with most functional languages, is easy if efficiency is unimportant and difficult otherwise. Chapter 1 pointed out that storage management and avoiding needless copies form a large part of the efficiency issue. This is exacerbated for **svexp** by the domain of problems where s-vals are handy; i.e., many interesting matrix problems are notoriously large; a single array may comprise nearly all of a program's storage requirements. Such problems are often deliberately scaled to squeeze into available memory resources.[1] To be taken seriously in this environment, a compiler cannot generate code that gratuitously makes copies of arrays,

[1]The author was reminded of this recently on overhearing a conversation of some scientific programmers. They were lamenting that a certain implementation of FORTRAN disallows array declarations larger than 512 megabytes. We look forward to the day when the cost of copying such a data structure is not important!

though this would be the propensity of a naive one.

5.1 The Standard Compilation Model

The usual approach to compilation of a language L is this:

- Define the target language T along with its semantics.

- Give a compiler as an inductively defined function $\mathcal{C}[\![\,\cdot\,]\!]$ that accepts a term from L (in the brackets $[\![\,]\!]$) along with n following arguments, which describe the form of compilation desired, and returns the compilation into T.

- Use the semantics of L and T to show in some desirable sense that for any program P and parameters $p_1, \ldots, p_n$ for the desired form of compilation, the meaning of $\mathcal{C}[\![P]\!]p_1 \cdots p_n$ in the semantics of T is equivalent to the meaning of L. This establishes the correctness of $\mathcal{C}$. There are several choices for the semantics, but a common one is denotational semantics for L and operational semantics for T.

- If the result from $\mathcal{C}$ is not efficient, we may define an additional function that transforms $\mathcal{C}[\![P]\!]p_1 \cdots p_n$ to another term in T that is more efficient. This is *optimization.*

Example 5.1 Let x range over names for integer storage locations and

$$T ::= \mathrm{load}\, x \mid \mathrm{store}\, x \mid \mathrm{add}\, x \mid T; T$$

with the usual semantics of an accumulator-based architecture. Suppose L has a binary $+$ operator. Then we might try the following for part of the compiler:

$$\mathcal{C}[\![E_1 + E_2]\!] = \mathcal{C}[\![E_1]\!];\ \mathrm{store}\, \mathit{tmp};\ \mathcal{C}[\![E_2]\!];\ \mathrm{add}\, \mathit{tmp}$$

and prove it correct by showing that if the code returned by $\mathcal{C}[\![E_1]\!]$ and $\mathcal{C}[\![E_2]\!]$ leave the correct values in the accumulator, then so will $\mathcal{C}[\![E_1 + E_2]\!]$. In the process we will learn that the value of *tmp* cannot be changed in whatever code is returned by $\mathcal{C}[\![E_1]\!]$ or $\mathcal{C}[\![E_2]\!]$, e.g. we should use a fresh temporary at every application of the rule. □

We will follow part of this compiler-building regimen, but depart from it in important ways. The first is the nature of $\mathcal{C}$. The code returned by $\mathcal{C}$ as described above can depend only on what has been "passed down" in the parameters $p_1 \cdots p_n$ (we have $n = 0$ in the example) and on the top level structure of the term to be compiled (+ in the example). The difficulty with this approach in our case is that the best compilation for a **svexp** subterm may depend on deep internal structure.

Example 5.2 Consider the term $\mathrm{Red} + 0\, E$. If E consists of several (typical) column vectors Join-ed together end to end, then we want to compile the reduction into several reducing loops whose results are added. The alternative is just one loop, which needs a tree of conditionals (hence run time for comparisons) in its body to accumulate a unique element from among the column vectors for each iteration. Yet writing a general compilation rule to express several loops is difficult because the structure of E is not apparent. □

There are several possible approaches to this problem, which appears in myriad forms in our compiler. We choose to define *code accumulations*. The language of code accumulations is just the target language T augmented with *active sites*. An active site has the form

$$[\![E, p_1, \ldots, p_n]\!] \tag{5.1}$$

where E is a subterm of L and $p_1 \cdots p_n$ are parameters that tell how to compile it, the same ones that would be passed as arguments in a traditional compiler.

Thus a code accumulation with active sites comprising some of its leaves is a partially compiled program. Specifically, the *rewrite* function $\mathcal{R}$ accepts a code accumulation, picks (arbitrarily) one of its active sites, and returns a rewritten code accumulation that is in some sense "more compiled" than the original.[2] Rewriting depends only on the top level structure of the chosen active site, but may entail the entire accumulation, in particular the path in the syntax tree from the active site to the root. The compiler $\mathcal{C}\langle\!\langle \cdot \rangle\!\rangle$ just applies $\mathcal{R}$ repeatedly until no active sites remain. We use a new kind of brackets $\langle\!\langle\ \rangle\!\rangle$ to remind the reader that the argument is a code accumulation, rather than an element of L. Let $(inactive E)$ return *true* when code accumulation E has no active sites, false otherwise. Then we have

$$\mathcal{C}\,\langle\!\langle P \rangle\!\rangle = inactive\,P \rightarrow \langle\!\langle P \rangle\!\rangle,\ \mathcal{C}(\mathcal{R}\,\langle\!\langle P \rangle\!\rangle) \tag{5.2}$$

Any traditional compiler can be expressed with code accumulations. Every compilation rule gives rise to a rewrite transformation which merely replaces an active site and does not change the rest of the accumulation. For instance, the rule in Example 5.1 yields

$$\mathcal{R}\langle\!\langle \xi[\![E_1 + E_2]\!] \rangle\!\rangle = \langle\!\langle \xi[\![E_1]\!]; \text{store } tmp; [\![E_2]\!]; \text{add } tmp \rangle\!\rangle \tag{5.3}$$

Here ξ denotes the *context* (ancestor nodes in the syntax tree) of the active site $[\![E_1 + E_2]\!]$.

Returning to Example 5.2, the compilation of Red+ $0\,E$ will initially be a reducing loop whose body is an active site containing E:[3]

$$\sum_{i=0}^{n-1} \langle\!\langle E, \ldots \rangle\!\rangle$$

where n is the inner length of the vector value of E obtained from its type. Thereafter, if E turns out to consist of Join'ed vectors, i.e. $E = \text{Join}\,0\,E_1\,E_2$, then $\mathcal{R}$ splits the

[2]The definition of "more compiled" involves the sizes of L-subterms in active sites. A simple count of active sites is not useful here because rewriting e.g. a binary operator application transforms one active to two.

[3]The actual target language uses a less standard but more convenient notation for summation ranges than that used here.

reducing loop into two loops:

$$(\sum_{i_1=0}^{n_1-1} \langle\!\langle E_1, \ldots \rangle\!\rangle) + (\sum_{i_2=n_1}^{n-1} \langle\!\langle E_2, \ldots \rangle\!\rangle)$$

where n_1 is the inner length of the value of E_1 taken from its type.

5.2 The Target Language

The target language for our compiler is called **proc** because its highest level construct is the *procedure.* It is given in Figure 5.1. Not surprisingly, the values in **proc** programs are arrays (not matrices). An s-val $V^{A;C}$ is represented in **proc** as an n-dimensional array with $n \leq |A| + |C|$. The inequality occurs when some axis is represented as the iterations of a loop. Consider for instance

$$f\, x\, y \equiv \mathrm{Proj}\, 7\, 0\, \mathrm{Red} +\ 1\, \mathrm{Typ}\, 0\, 7\, (\mathrm{Join}\, 1\, x\, y)$$

which, given x and y are column vectors, produces their vector sum. The value of $(\mathrm{Join}\, 1\, x\, y)$ has $|A| + |C| = 0 + 2 = 2$. Yet only 1-dimensional arrays, for x, y, and the result value, appear in the compilation; the iterations of a summation effectively represent the missing dimension. We compile each k-ary **svexp** function into a procedure of $k + 1$ arguments; the extra one is a reference to a result array. Hence, the **svexp** program body compiles to a procedure of one argument, the array for the program's answer. For convenience, we adopt the convention that a **svexp** function and its compilation as a procedure have the same name.

The **proc** procedure named f, obtained by compiling the **svexp** user function definition for f, has two useful interpretations. The first is an *imperative procedure* that accepts arrays $a_1, \ldots, a_n$ as arguments and stores a result into array r, as in (5.4). In this case, we interpret the $=$ in (5.5) as an assignment operator; thus the term may be executed by a computer. This is simply the compilation we are after. The second interpretation is a *declaration* of the value of r in terms of the array values a_k. Here

$$\langle proc \rangle ::= \langle proc\text{-}name \rangle(a_1, \ldots, a_n;\ r) \rhd \langle assign \rangle \tag{5.4}$$

$$\langle assign \rangle ::= \langle subarray \rangle = \langle expr \rangle \mid \tag{5.5}$$

$$\langle proc\text{-}name \rangle(\langle subarray \rangle^*;\ \langle subarray \rangle) \mid \tag{5.6}$$

$$\bigvee_{k \in \langle rng \rangle} \langle assign \rangle \mid \tag{5.7}$$

$$\langle assign \rangle \wedge \langle assign \rangle \mid \tag{5.8}$$

$$\text{if } \langle expr \rangle \text{ then } \langle assign \rangle \text{ else } \langle assign \rangle \mid \tag{5.9}$$

$$\text{let } \langle array\text{-}id \rangle \rhd \langle assign \rangle \text{ in } \langle assign \rangle \tag{5.10}$$

$$\langle expr \rangle ::= \langle subarray \rangle \mid \tag{5.11}$$

$$\sum_{k \in \langle rng \rangle} \langle expr \rangle \mid \tag{5.12}$$

$$\langle expr \rangle + \langle expr \rangle \mid \tag{5.13}$$

$$\text{if } \langle expr \rangle \text{ then } \langle expr \rangle \text{ else } \langle expr \rangle \mid \tag{5.14}$$

$$\text{let } \langle array\text{-}id \rangle \rhd \langle assign \rangle \text{ in } \langle expr \rangle \tag{5.15}$$

$$\langle rng \rangle ::= lo{:}size \tag{5.16}$$

$$\langle subarray \rangle ::= [\langle perm \rangle]\langle array\text{-}id \rangle[\langle indexer \rangle^*] \tag{5.17}$$

$$\langle indexer \rangle ::= \langle integer \rangle \mid \langle rng \rangle \tag{5.18}$$

$$\langle perm \rangle ::= \langle integer \rangle^* \tag{5.19}$$

Figure 5.1: The target language **proc**.

we consider the = in (5.5) to be a declaration of equality. For this interpretation to be valid, the compilation must define exactly one value for each element of array r (and any intermediate results defined by *let*); otherwise the declaration is contradictory or incomplete. The corresponding effect in the imperative interpretation is that each array element in r (and in *let*-introduced temporaries) is assigned exactly one value during the procedure's execution.

With this *single assignment* property, it is not hard to see that the interpretations are related in an important way: If we can show that the declarative interpretation matches the semantic value of f under some one-to-one relation between arrays and matrices, then we know that the imperative interpretation always produces the correct answer; the **svexp** and **proc** programs are just different ways of writing the same function. On the other hand, the "if" above is a rather big one. It means we must develop a way to reason about the equivalence of semantic values of subterms of a given **svexp** function and the declarative interpretation if its compilation.

5.2.1 Semantics of proc

Let us describe **proc** programs in more detail. Rule (5.4) declares the procedure formal parameters and result. Valid procedures use parameters a_k only on the right hand side of =, and r only on the left hand side. In the imperative interpretation, then, parameters a_k may be passed by reference (i.e. as pointers), and the storage for the result may be allocated by the caller and also passed by reference.

Array variables in **proc** are bound to arrays with a recursive structure, i.e. an $n{+}1$-dimensional array is an array of n-dimensional ones.

We consider an n-dimensional array to have n axes, just as matrices. The essential difference is that the axes of an n-dimensional array are always numbered zero through $n-1$. We write $n(a)$ for the number of axes of the array represented by a and $d_k(a)$ with $k \geq 0$ for the bound on the k-th dimension of a. We also write

$a : d_0 \times \cdots \times d_{n(a)-1}$ to assert the structure of a. To reference an array element, we use brackets, i.e. $a[i_0, \ldots, i_{n(a)-1}]$, where the i_k range over natural numbers. Also consistent with our matrix terminology, we speak of index element i_k interchangeably as occurring in the k-th index position or the k-th axis of a. An array a with $n(a) = 0$ contains only one value. We adopt the common practice of identifying 0-dimensional arrays with the values they contain. Thus $a + a$ makes sense; we are adding scalar values to get a new scalar.

Rule (5.5) asserts the equality (resp. assigns the value of) the expression on the right hand side of $=$ and the array reference. The $\langle subarray \rangle$ is restricted to be a 0-dimensional one; only one array element's value is defined (resp. assigned) at a time.

Rule (5.6) describes an assertion of the form

$$f(A_1, \ldots, A_n; R)$$

where A_i and R are subarrays. It declares that R has the value assigned by the body of the declaration of procedure f as in (5.4) when its parameters $a_1, \ldots, a_n$ have values $A_1, \ldots, A_n$ respectively.

The imperative interpretation is a procedure call, where subarrays are passed both as arguments and for the return value. In contrast with the simple assignment of (5.5), a procedure call can effectively assign many scalar values with no $\forall$, as in (5.7). Call-by-reference semantics are assumed for all array passing. Hence the compiler must allocate (with a *let*) fresh memory for arguments and return values whenever these are not subarrays of the parameters or result of the procedure in which the call occurs.

Rule (5.7) asserts that the included equality holds (resp. iterates the included assignment) over a range of subscripts.

Rule (5.8) asserts the equalities on both sides of the $\wedge$ (resp. executes both assignments in parallel). The left and right hand sides may both *use* any given value, but, due to the single assignment property, they cannot both *define* (resp. assign) the same array element.

Rule (5.9) conditionally asserts the equality (resp. evaluates the assignment) of either its *then* or *else* branch based on the value of its boolean predicate.

Rule (5.10) defines an intermediate value (resp. declares a temporary array) a with its first $\langle assign \rangle$ term, then asserts (resp. evaluates) the second $\langle assign \rangle$, which may refer to a on the right hand side of $=$. Since no **proc** code outside the let can properly refer to a, its storage may be freed when execution of the second $\langle assign \rangle$ is complete.

Rule (5.11) states that a subarray is a valid expression, as is a vector reduction (5.12) and scalar binary + (5.13).

Rule (5.14) conditionally returns the value of either its *then* or *else* branch based on the value of its boolean predicate. The compiler generates only *ifs* with 0-dimensional values.

Rule (5.15) is the same as (5.10), except it returns the value of the expression in its *in* part. As a practical matter, the compiler restricts the value to be the result of $\sum$, +, or *if*, which are all 0-dimensional. Hence a stack suffices for allocating the storage for *let*-defined arrays in **proc** programs.

Rule (5.16) gives a uniform syntax for index ranges in **proc**. The range $lo{:}size$ contains the natural numbers from lo to $lo + size - 1$ inclusive. Thus $k \in lo{:}size$ is equivalent to $lo \leq k < lo{+}size$. The names lo and $size$ represent natural number expressions, which have the same syntax as $\langle dim \rangle$ in Figure 4.11.

Rule (5.17) describes subarrays, which are the subject of the next section.

5.2.2 Subarrays

We consider first the [⟨*indexers*⟩] part of the subarray syntax, then the [⟨*perm*⟩] part. A subarray of a must have $n(a)$ indexers as described in rule (5.18). A *simple indexer* e is just an expression over natural numbers; it selects a single component of a. A *range indexer* selects a range of components. The following is a direct consequence: Let a be an array and b be a subarray of a with s simple indexers and r range indexers. Then we have $n(b) = r = n(a) - s$.

Example 5.3 Suppose $a : 3 \times 4 \times 5$ and let $b = a[1{:}2, 3, 3{:}1]$. Then $n(b) = 2$; and we have $b : 2 \times 1$. Let $c = b[1, 0{:}1]$. Then $n(c) = 1$, and $c : 1$ (i.e. c is a vector of one element). Furthermore $c = a[2, 3, 0{:}1]$. Finally, $d = c[0]$ is 0-dimensional, and $d[\,] = d$. □

In general, $a[0{:}d_0(1), \ldots, 0 : d_{n(a)-1}(a)] = a$. Thus if a is 0-dimensional, we have $a[\,] = a$. This is consistent with 0-dimensional arrays as values as discussed on page 119.

We now consider the effect of the [⟨*perm*⟩] part of a subarray of array a, which is a permutation of the range $0{:}n(a)$. If $[p_0, \ldots, p_{n(a)-1}]$ is such a permutation, then

$$a' = [p_0, \ldots, p_{n(a)-1}]a \tag{5.20}$$

is also an array related to a by

$$a'[i_0, \ldots, i_{n(a)-1}] = a[i_{p_0}, \ldots, i_{p_{n(a)-1}}] \tag{5.21}$$

I.e. the permutation specifies a generalized transposition. If $p_k = k$ for $0 \leq k \leq n(a){-}1$, then no transposition occurs. For convenience, we can omit the [⟨*perm*⟩] part of the subarray notation when this is the case. Thus the array element reference notation $a[\cdot]$ we have been using is just a subarray with all indexers simple and no transposition.

Example 5.4 Consider once more the dot product. Assume that a_1 and a_2 are 1-dimensional arrays with $d_0(a_1) = d_0(a_2) = d$. Then we have

$$dot(a_1 a_2;\ r)\ \rhd r[\,] = \sum_{k \in 0:d} (a_1[k] \times a_2[k])$$

and for the transpose of matrix multiply $(a_1\, a_2)^T$, we have:

$$mmt(a_1 a_2;\ r)\ \rhd \underset{i \in 0:d_0}{\forall}\ \underset{j \in 0:d_2}{\forall}\ dot(a_1[i, 0{:}d_1],\ a_2[0{:}d_1, j];\ [1,0]r[i,j]) \tag{5.22}$$

with the understanding that $a_1 : d_0 \times d_1$ and $a_2 : d_1 \times d_2$. Note how the transposition $[1, 0]$ is applied at the dot product result. □

5.3 Compiling proc to C

The reader may properly raise an eyebrow at the definition of **proc**, in that some of its features—subarrays and transposition, for instance—apparently require temporary space and copying to implement. Hence before undertaking the task of translating **svexp** to **proc**, we consider the simpler job of showing that **proc** can indeed be translated to machine code that is quite efficient.

5.3.1 Dope vectors

Dope vector is a classical term for a run-time structure that encodes referencing information for an array, typically the sizes of each dimension. Some form of dope vector is required to represent an array (in addition to a block of storage for its elements) whenever the array size is not known at compile time.

In most language implementations the dope vector for n-dimensional array a is really a record. It contains a pointer $a.p$ that points to the start of the element storage block and a vector $a.d$ of n integers. The vector elements store the size of each array axis. These may be used for "range checking" index values as well as to reference array elements. For the latter, a typical convention to find the address of element $a[k_0, k_1, \ldots, k_{d-1}]$ is:

$$a.p + k_0 + \sum_{i=1}^{d-1} \left(k_i \prod_{j=0}^{i-1} a.d_j \right)$$

This effectively interprets indices as a lexicographic order, with elements farther to the right receiving higher weight, a so-called column-major order. This is a suitable practice for languages where transposition is not a primitive. However, it is not sufficient for **proc**, where it would require copying to effect the axis permutation of (5.17).

Hence we generalize dope vectors to achieve the following features:

- An array may be transposed by manipulating only its dope vector.
- A subarray may be formed by creating only a new dope vector whose pointer points into the existing block.
- Given the dope vector and the address of any array element a, the address of any element adjacent to a (along any axis) may be computed with one addition.

Fortunately, there is little overhead for this generality. The dope vector for an n-dimensional array a in our case has this structure:

- Pointer $a.p$. This points to the first location of the element storage block as before.
- Stride vector $a.s$ with n elements; $a.s_k$ gives the spacing between elements along axis k.
- Size vector $a.d$ with n elements. Element $a.d_k$ gives the size of the k-th axis.

5.3.2 Element addresses and transposition

Rectangularity constrains the values of $a.s$. The constraints are founded in an implicit permutation of the axis numbers $0 \leq k < n(a)$. Let p_k be the k-th value of this permutation. Then if a is an array in contiguous storage (i.e. it is not a general subarray as described below), we have

$$a.s_{p_k} = \prod_{j=0}^{k-1} a.d_{p_j} \quad \text{for} \quad 0 \le k < n(a).$$

From this it is easy to see that our generalized dope vector encodes a lexicographic order on index elements just as the conventional one, though here the order is fixed only at run time. The element address formula is

$$a.p + \sum_{k=0}^{d-1} i_k \cdot a.s_k \tag{5.23}$$

5.3.3 Subarrays

Consider the **proc** subarray expression

$$[p_0, \ldots, p_{n(a)-1}]a[I_0, \ldots, I_{n(a)-1}]$$

as in (5.20) with a represented as a dope vector. Then the value above is just another dope vector a'. Let R be the subsequence of $[I_0, \ldots, I_{n(a)-1}]$ consisting of range indexers $I_k = lo_k{:}size_k$; the rest are simple indexers. Extend lo so that $lo_k = I_k$ if I_k is simple. Let $[r_0, \ldots, r_{|R|-1}]$ be the subsequence of $[p_0, \ldots, p_{n(a)-1}]$ such that $I_{p_k} \in R$. Then

$$\begin{aligned} a'.p &= a.p + \sum_{k=0}^{n(a)-1} lo_{p_k} \cdot a.s_k \\ a'.s_k &= a.s_{r_k} \qquad \text{for } 0 \le k < |R| \\ a'.d_k &= a.size_{r_k} \quad \text{for } 0 \le k < |R| \end{aligned} \tag{5.24}$$

Simple algebra shows that (5.23) applied to the subarray thus constructed addresses elements as described for rule (5.17) above.

Most importantly, the construction of a' involves no copying of the storage block of a into fresh storage. It requires only $O(n(a))$ time and space.

Table 5.1: Dope vector *a1* as parameters.

Dope vector	C translation
$a1.p$	a1
$a1.s_0$	a1s0
$a1.s_1$	a1s1
$a1.d_0$	d4
$a1.d_1$	d3

5.3.4 Strength-reduced stepping

We have shown that copying can be avoided in translating **proc** to machine code, but there are still some nagging expenses. Consider the result subarray $r' = [1,0]r[i,j]$ passed to *dot* in (5.22). This involves the computation

$$r'.p = r.p + i \cdot r.s_1 + j \cdot r.s_0$$

for each call to *dot*. This is expensive on most machines due to the multiplications. If in practice the $\forall$ loops are executed sequentially, efficient machine code will compute the address of $[1,0]r[i,j]$ initially for $i = j = 0$, then update it by addition for successive iterations. In particular, incrementing i is equivalent to adding $r.s_1$ to the previous address; incrementing j is the same as $r.s_0$. This transformation is just a composition of two well-known optimizations—induction variable elimination and strength-reduction [ASU86]. Experience such as [Mey88] shows that it speeds array computations considerably. Yet it is non-trivial to apply in general imperative languages because the structure of iterations must be gleaned by the compiler; **proc** presents no such problems. For illustration, Figure 5.2 shows a compilation to C of *dot* and *mmt*. Dope vectors are represented in parameter lists of C functions. For example, the dope vector for *a1*, the first parameter to *mmt* is represented as in Table 5.1. The names d4 and d3 were created by the dimension inference algorithm of the compiler implementation.

```
void dot(a1, a1s1, d1, a2, a2s0, d2, r)
FLOAT_PTR a1; int a1s1, d1;
FLOAT_PTR a2; int a2s0, d2;
FLOAT_PTR r;
{
 FLOAT_PTR a2p0, a1p1;
 FLOAT t0;
 int i0;

 t0 = 0;
 for (a2p0 = a2, a1p1 = a1, i0 = 0;
      i0 < d1;
      a2p0 += a2s0, a1p1 += a1s1, ++i0)
  t0 += (*a1p1) * (*a2p0);
 *r = t0;
}

void mmt(a1, a1s0, d4, a1s1, d3, a2, a2s0, d5, a2s1, d6, r, rs0, rs1)
FLOAT_PTR a1; int a1s0, d4, a1s1, d3;
FLOAT_PTR a2; int a2s0, d5, a2s1, d6;
FLOAT_PTR r; int rs0, rs1;
{
 FLOAT_PTR rp0, rp1, a2p2, a1p3;
 int i0, i1;

 for (rp0 = r, a1p3 = a1, i0 = 0;
      i0 < d4;
      rp0 += rs1, a1p3 += a1s0, ++i0)
  for (rp1 = rp0, a2p2 = a2, i1 = 0;
       i1 < d6;
       rp1 += rs0, a2p2 += a2s1, ++i1)
   dot(a1p3, a1s1, d3, a2p2, a2s0, d5, rp1);
}
```

Figure 5.2: Example of **proc** translation to C.

5.4 A Correctness Criterion

We now undertake the task of compiling **svexp** user function definitions to **proc**.[4] The first step is to form the initial code accumulation to pass to the compiler $\mathcal{C}$ as in (5.2). For a function definition

$$f\,x_1\,\ldots\,x_n \equiv E : C, \gamma \tag{5.25}$$

where $E : C, \gamma$ denotes that type inference yields an inner map that denotes axis set C and dimension function γ for the function body, the corresponding accumulation is

$$\langle\!\langle f(a_1, \ldots, a_n;\; r) \;\triangleright [\![E, =, r, 0_{0:|C|}, 0_C, \lambda c.\langle (q_C\, c), (\gamma\, c)\rangle, \emptyset]\!]\rangle\!\rangle \tag{5.26}$$

This is a just a procedure whose body is an active site containing E, the function body from (5.25). The extra objects in the active site are parameters to direct the rewrite function as in (5.1). We discuss them in detail as we give $\mathcal{R}$ below.

A correct compilation of a **svexp** function named f to a **proc** procedure named f can now be characterized intuitively in terms of a one-to-one "translation" between matrices and arrays as follows.

- Consider the semantic value f bound to the name f in Equation (3.15).
- Consider matrices $M_1, \ldots, M_n$ of suitable type to be arguments of f. Translate them to arrays $a_1, \ldots, a_n$ and consider these to be the values of the parameters of procedure f.
- Determine the array value r asserted by the fully compiled body of (5.26).
- Procedure f is correct if the translation of the matrix

  $$f\,M_1\,\ldots\,M_n$$

  to an array value is equal to r for all $M_1, \ldots, M_n$ of suitable type.

[4]The "def" body of a **svexp** program is just a nullary function application, so we will not discuss it as a separate case.

Formalization of this idea begins with the notion of "translation."

Definition 5.5 *Suppose $M^{A,\alpha}$ and let $n = |A|$. Also let $q : A \to (0{:}n)$ be a 1-1* axis map *(thus q^{-1} exists). The* translation of M in **proc**, *denoted $(\mathbf{T}\, M\, q)$, is the n-dimensional array $a : \alpha\,(q^{-1}\, 0) \times \cdots \times \alpha\,(q^{-1}\,(n-1))$ such that*

$$a[i_0, \ldots, i_{n-1}] = M\, \mathbf{0}\, [i_0/(q^{-1}\, 0)] \ldots\ [i_{n-1}/(q^{-1}\,(n{-}1))] \qquad (5.27)$$

Similarly, if we are given array a and inverse axis map q^{-1}, then (5.27) also describes the translation of a in **svexp**, *denoted $(\mathbf{T}^{-1}\, a\, q^{-1})$.*

Finally, if q is constrained to be monotonic with respect to the order of natural numbers, then it is unique for a given axis set A. We call this the canonical axis map *for A and denote it q_A. The translations it induces are* canonical translations. *The canonical translation of M^A in* **proc** *is thus $(\mathbf{T}\, M\, q_a)$, which we shorten to $(\mathbf{T}\, M)$ for convenience. Inversely, the canonical translation of a in* **svexp** *is $(\mathbf{T}^{-1}\, a\, q_A^{-1})$, which we can shorten to $(\mathbf{T}^{-1}\, a\, A)$, since A determines q_A^{-1} uniquely.* □

Example 5.6 Suppose $M^{A,\alpha}$ where $A = \{2, 5\}$, $\alpha\, 2 = 3$ and $\alpha\, 5 = 4$. The canonical map q_A has $q_A\, 2 = 3$ and $q_A\, 5 = 4$. Thus $(\mathbf{T}\, M) : 3 \times 4$. □

With this, we are ready to give a precise notion of what it means for a procedure to be correct.

Definition 5.7 *Let ϕ be a* **svexp** *function variable environment wherein f is bound to an n-ary function. Let procedure f be a* **proc** *procedure that results from compiling the definition (5.25). Recall that a procedure call $f(E_1, \ldots, E_n; E)$ is an assertion of the value of E in terms of the values $E_1, \ldots, E_n$.*

We say procedure f is correct with respect to ϕ *if for all matrices $M_1, \ldots, M_n$ the following holds:*

$$\begin{array}{ll} M = (\phi f)\, M_1\, \ldots\, M_n \notin \mathbf{W}_t & \textit{(for } \textbf{svexp} \textit{ function } f\textit{) implies} \\ f((\mathbf{T}M_1), \ldots, (\mathbf{T}M_n);\ (\mathbf{T}M)) & \textit{(for } \textbf{proc} \textit{ procedure } f\textit{)} \end{array} \qquad (5.28)$$

□

That is, all calls to procedure f on the translations of $M_1, \ldots, M_n$ assert result values that match the return value of **svexp** function f ' in ϕ on $M_1, \ldots, M_n$, whenever that return value is not $\perp$ or a type error.

Rule (5.4) states that the form of procedure f is

$$f(a_1, \ldots, a_n;\ r) \rhd B \tag{5.29}$$

It is not hard to show that a sufficient condition for procedure f's correctness with respect to ϕ is the following property of its body.

Property 5 *For any matrices $M_1, \ldots, M_n$, let*

$$\beta = \perp [\lambda i.M_1/x_1] \cdots [\lambda i.M_n/x_n]$$

and in B of (5.29) let

$$a_1 = (\mathbf{T}\, M_1), \ldots, a_n = (\mathbf{T}\, M_n)$$

If $V = \mathcal{E}[\![E]\!]\,\phi\beta \notin \mathbf{W}_t$, then B asserts that r has array value $\mathbf{T}\,(V\,\mathbf{0})$. $\Box$

Thus our approach to compiling correct procedures is this:

- Extend $\mathbf{T}$ to active sites, where it returns a fragment of **proc**.[5]
- Extend $\mathbf{T}$ again to entire code accumulations. A translated accumulation is the same as the original except that all active sites are replaced with their translations.
- Show that the translation of (5.26) is correct by verifying Property 5.
- Derive $\mathcal{R}$ so that if ξ is a code accumulation with an active site and $(\mathbf{T}\,\xi)$ is correct, then $\mathbf{T}\,(\mathcal{R}\,\xi)$ is correct.

[5]In fact it is notationally simpler to use hybrid fragments that are part **proc** and part **svexp** semantic expression, and this is the course we take.

It is important to stress that **T** and the value it returns are ephemeral tools for establishing the correctness of compilations in progress. Though **T** returns **proc** code, **T** is not part of the compiler, and its return value is useful only for its declarative interpretation, which stands for the semantic value of an uncompiled fragment of **svexp**.

5.4.1 Active sites

The active sites in Example 5.3 are simple, containing only a fragment of source code to compile. As shown in (5.26), however, **svexp** is more complex. Active sites must contain enough information to tell the rewrite function $\mathcal{R}$ *how* to go about rewriting. In particular, this information much be rich enough so that $\mathcal{R}$ can make good decisions on where to emit two key structures:

- The *let*s which allocate storage resources at run time.
- Loops ($\forall$, $\sum$ and $\prod$) that, in a manner of speaking, allocate computation resources and generally determine the asymptotic running time.

In this case, procrastination is a virtue. The best course is always to delay emitting lets and loops as long as possible, until it is certain they are needed, and then to emit them in the best possible place. For instance, we generally want lets as far outward in the nest of loops as possible, so that values are not recalculated unnecessarily in the imperative interpretation of the result.

There are two classes of active sites—those that are rewritten into **proc** $\langle$*assign*$\rangle$ subterms and those that are rewritten into $\langle$*expr*$\rangle$s. We consider these in order.

5.4.2 Active Assignments

The general form of an *active assignment site* is

$$[\![E, =, tgt, \textit{tgt-base}, \textit{val-base}, \textit{size}, \textit{i-vars}]\!] \tag{5.30}$$

To understand the components of this structure, it is helpful first to extend the notion of range indexers of subarrays backward into semantic domains—to matrices. We did this in a specialized way by extracting all but the first row in the semantic equation for Extr (3.4). The generalization is straightforward; the submatrix of M from lo to $lo + size - 1$ along axis a is just

$$\lambda i\,.\,(i\,a) < size \rightarrow M\,i\,[(i\,a) + lo/a], rangerr$$

This notion extends naturally to s-vals; we can extract subranges along outer axes as well as inner ones to obtain *sub-s-vals.*

Another term: we say that a loop is *ancestral* to an active site if the active site is within the loop body.

Then the components of (5.30) have commonsense meanings as follows.

E The **svexp** subterm to be compiled.

= A flag denoting that this is an active assignment. Active expression sites have ? instead.

tgt The array, called the *target*, into which assignment is to be performed. In fact, the assignment is to a subarray of *tgt* is in (5.5). The rest of the subarray is encoded in other fields.

tgt-base Encodes the values lo_k as in (5.24) of the target subarray.

val-base Encodes lo values for a sub-s-val of the semantic value of E. This gives the values assigned to *tgt* and is called the *source.*

size Encodes *size* values for *both* the target subarray and the source sub-s-val, but includes only axes for which ancestral loops do not already exist.

i-vars Encodes loops ancestral to the active site.

It remains to note the form of all these "encodings." They are simply maps that take axis numbers to other values—*tgt-base* takes target array axes to symbolic arithmetic expressions derived from types, and *val-base* takes s-val axes to similar expressions. The map *size* takes s-val axes to pairs of the form $\langle$*array axis*, *arithmetic expression*$\rangle$. Finally, *i-vars* takes s-val axes to the names of loop variables, which conveniently serve as identifiers for the loops themselves. While the inputs and outputs of user functions are matrices, intermediate values in the function body are s-vals. Thus the domains of *val-base*, *size*, and *i-vars* may include both inner and outer axis numbers. For this reason, we assume that axis numbers are "tagged" so that same-numbered inner and outer axes are distinct.

Such maps are easily implemented as a sequence of pairs; e.g. patching and application modify and search the sequence respectively. The shorthand $(size_1\, a)$ denotes the array axis part of the pair mapped from a by *size*. Similarly $(size_2\, a)$ gives the expression part. The domain of a map m is $\mathrm{dom}\, m$, and $\mathrm{ran}\, m$ is its range. Other operations are introduced as required.

We are now in a position to describe the contents of the active assignment in (5.26):

r The target array, which is the result value.

$0_{0:|C|}$ (*tgt-base*) A map taking the axes of the target array to zero. These are the *lo* values as in (5.24) of the target subarray.

0_C (*val-base*) A map taking axes of the function body s-val to zero. These are the *lo* values as in (5.24) of the source sub-s-sval.

$\lambda c.\langle(q_C c), (\gamma\, c)\rangle$ A map taking every s-val axis of the function body to a pair consisting of the corresponding canonical array axis and the function result size obtained by type inference.

$\emptyset$ The empty map of loops already emitted.

The translation of an active assignment

$$\mathbf{T}\,[\![E, =, tgt, tgt\text{-}base, val\text{-}base, size, i\text{-}vars]\!] \tag{5.31}$$

is based first on the idea that *size* represents $\forall$ loops necessary to fill in the target. Thus we construct a loop nest with one loop for each element of the range of *size*. Correct source values come from the obvious place—the semantic value of E. Let ϕ and β be environments as assumed in Property 5, page 129. Recall that $\operatorname{ran} m$ is the range of map m. Suppose $\operatorname{ran} size = \{\langle b_1, e_1\rangle, \ldots, \langle b_m, e_m\rangle\}$ and $V^{A;C} = \mathcal{E}[\![E]\!]\phi\,\beta$ where $A = \{a_1, \ldots, a_p\}$ and $C = \{c_1, \ldots, c_q\}$. Finally let the target be n-dimensional. Then the desired translation (5.31) is

$$\begin{array}{c} \underset{i_{b_1}\in 0:e_1}{\forall} \cdots \underset{i_{b_m}\in 0:e_m}{\forall} tgt[i_0 + (tgt\text{-}base\,0), \ldots, i_{n-1} + (tgt\text{-}base\,(n-1))] = \\ V\ \mathbf{0}\,[(M\,a_1)/a_1]\cdots[(M\,a_p)/a_p]\ \ \mathbf{0}\,[(M\,c_1)/c_1]\cdots[(M\,c_q)/c_q] \end{array} \tag{5.32}$$

where with a an s-val axis (inner or outer)

$$M\,a = \begin{cases} i_b + (val\text{-}base\,a) & \text{if } (size_1\,a) = b \\ (i\text{-}vars\,a) + (val\text{-}base\,a) & \text{if } a \in (\operatorname{dom} i\text{-}vars) \\ (val\text{-}base\,a) & \text{otherwise} \end{cases} \tag{5.33}$$

We have adopted the convention of using i_b for the $\forall$ variable to iterate over axis b of the target, whether that variable is code already emitted or in a translation (5.32). Loop variables i_b on the left side of the equality that are not i_{b_k} for any k (i.e. $b \notin (\operatorname{ran} size_1)$) must be in ($\operatorname{ran}$ *i-vars*). That is, they are variables of ancestral loops. This relationship must be preserved by $\mathcal{R}$. Similarly, the cases in (5.33) must be disjoint and well-defined.

We can now readily verify that the translation of the initial code accumulation in (5.26) is correct. Here we have $(\operatorname{dom} size) = \{c_1, \ldots, c_n\} = C$, where C is the axis set of the function result. Assume without loss of generality that $c_1 < c_2 < \cdots < c_n$.

Then the translation is

$$\underset{i_0 \in 0:(\gamma\, c_1)}{\forall} \cdots \underset{i_{n-1} \in 0:(\gamma\, c_n)}{\forall} r[i_0, \ldots, i_{n-1}] = V\, \mathbf{0}\, \mathbf{0}\, [i_0/c_1] \cdots [i_{n-1}/c_n] \tag{5.34}$$

which satisfies Property 5 (page 129) by inspection.

5.4.3 Active Expressions

The general form of an active expression site is this:

$$[\![E, ?, \textit{val-base}, \textit{i-vars}]\!]$$

The ? is just a flag to denote that this is an active expression; *val-base* and *i-vars* serve the same purpose as in active assignments. An active expression is, roughly speaking, the source value of an active assignment. Thus the absence of a *tgt* field is expected. In addition, there is no *size* field because the translation of an active expression is always a 0-dimensional value. Thus the translation is a subarray with simple indexers and a trivial permutation part. Let V be as above; then

$$\begin{aligned} &\mathbf{T}\, [\![E, ?, \textit{val-base}, \textit{i-vars}]\!] = \\ &\quad V\ \mathbf{0}\, [(M\, a_1)/a_1] \cdots [(M\, a_p)/a_p]\ \mathbf{0}\, [(M\, c_1)/c_1] \cdots [(M\, c_q)/c_q] \end{aligned} \tag{5.35}$$

where

$$M\, a = \begin{cases} (\textit{i-vars}\, a) + (\textit{val-base}\, a) & \text{if } a \in (\text{dom } \textit{i-vars}) \\ (\textit{val-base}\, a) & \text{otherwise} \end{cases}$$

5.5 The Rewrite Function

The machinery is now in place to derive $\mathcal{R}$ so that it maintains correctness. Most of the remaining work is to keep track of details as derivation proceeds by straightforward algebraic reasoning.

5.5.1 Rewriting active assignments

Proj. Consider the code accumulation

$$\langle\!\langle \xi [\![\text{Proj}\ a\ c\ E, =, \textit{tgt}, \textit{tgt-base}, \textit{val-base}, \textit{size}, \textit{i-vars}]\!] \rangle\!\rangle \tag{5.36}$$

Let $V^{A;C} = \mathcal{E}[\![\mathrm{Proj}\, a\, c\, E]\!]\,\phi\beta = \lambda ij\,.\,V'\,i\,[(j\,c)/a]\,j$, where ϕ and β are as in Property 5, page 129, and $V'^{A';C'} = \mathcal{E}[\![E]\!]\,\phi\beta$. The translation of the active site is then

$$\begin{aligned}
&\underset{i_{b_1}\in 0:e_1}{\forall} \cdots \underset{i_{b_m}\in 0:e_m}{\forall}\ tgt[i_0 + (\mathit{tgt\text{-}base}\,0), \ldots, i_{n-1} + (\mathit{tgt\text{-}base}\,(n-1))]\\
&\quad = (\lambda ij\,.\,V'\,i\,[(j\,v)/a])\,\mathbf{0}\,[(M\ a_1)/a_1]\cdots[(M\ a_p)/a_p]\,\mathbf{0}\,[(M\ c_1)/c_1]\cdots[(M\ c_q)/c_q]\\
&\quad = V'\,\mathbf{0}\,[(M\ a_1)/a_1]\cdots[(M\ a_p)/a_p]\,[(M\ c)/a]\,\mathbf{0}\,[(M\ c_1)/c_1]\cdots[(M\ c_q)/c_q]
\end{aligned} \tag{5.37}$$

with all the relevant auxiliary definitions from (5.32). Our goal is to show that this is the translation of some active assignment that is an easily computed transformation of (5.36). Chapter 3 showed Proj to be merely a renaming of outer axis a of the operand to make it inner axis c in the result. Thinking about this relationship in reverse—operand axis a as renamed result axis c—leads us to try

$$[\![E, =, \mathit{tgt}, \mathit{tgt\text{-}base}, (\mathit{remap}\ \mathit{val\text{-}base}\, c\, a), (\mathit{remap}\ \mathit{size}\ c\, a), (\mathit{remap}\ \mathit{i\text{-}vars}\ c\, a)]\!] \tag{5.38}$$

where $f' = (\mathit{remap}\ f\ x\ y)$ is the same as f except $f'\,y = f\,x$ and $f'\,x$ is undefined. We know from Theorem 3.8 that

$$c \notin C',\ a \notin A,\ C = C' \cup \{c\},\ \text{and}\ A' = A \cup \{a\} \tag{5.39}$$

With these it is a substitution exercise to show that the translation of (5.38) is equal to (5.37). Thus the desired rewrite equation is

$$\begin{aligned}
&\mathcal{R}\langle\!\langle \xi[\![\mathrm{Proj}\ a\, c\, E, =, \mathit{tgt}, \mathit{tgt\text{-}base}, \mathit{val\text{-}base}, \mathit{size}, \mathit{i\text{-}vars}]\!]\rangle\!\rangle\\
&\quad = \langle\!\langle \xi[\![E, =, \mathit{tgt}, \mathit{tgt\text{-}base}, (\mathit{remap}\ \mathit{val\text{-}base}\, c\, a), (\mathit{remap}\ \mathit{size}\ c\, a), (\mathit{remap}\ \mathit{i\text{-}vars}\ c\, a)]\!]\rangle\!\rangle
\end{aligned}$$

Typ. It is tempting to conclude immediately that

$$\begin{aligned}
&\mathcal{R}\langle\!\langle \xi[\![\mathrm{Typ}\ c\, a\, E, =, \mathit{tgt}, \mathit{tgt\text{-}base}, \mathit{val\text{-}base}, \mathit{size}, \mathit{i\text{-}vars}]\!]\rangle\!\rangle\\
&\quad = \langle\!\langle \xi[\![E, =, \mathit{tgt}, \mathit{tgt\text{-}base}, (\mathit{remap}\ \mathit{val\text{-}base}\, a\, c), (\mathit{remap}\ \mathit{size}\ a\, c), (\mathit{remap}\ \mathit{i\text{-}vars}\ a\, c)]\!]\rangle\!\rangle
\end{aligned} \tag{5.40}$$

because Proj often acts as the inverse of Typ; but, it is not exactly an inverse. If we construct a translation similar to (5.37) for Typ $c\,a\,E$, then (5.40) is correct (again by simple substitution) only if

$$c \notin C,\ a \notin A',\ C' = C \cup \{c\},\ \text{and } A = A' \cup \{a\} \tag{5.41}$$

This is not always the case, however. The following function computes the determinant of a triangular matrix *xt*.

$$\mathit{tdet}\ \mathit{xt} = \mathrm{Red}{*}\,0\ \mathrm{Proj}\,7\,0\ \mathrm{Typ}\,0\,7\ \mathrm{Typ}\,1\,7\ \mathit{xt}$$

At some point in compiling *tdet*, the code accumulation is of the form

$$\langle\!\langle \xi[\![\mathrm{Typ}\,0\,7\ \mathrm{Typ}\,1\,7\,\mathit{xt}, \ldots]\!]\rangle\!\rangle$$

Hence, $V'^{A'} = \mathcal{E}[\![\mathrm{Typ}\,1\,7\,\mathit{xt}]\!]\,\phi\beta$ and $V = \mathcal{E}[\![\mathrm{Typ}\,0\,7\ \mathrm{Typ}\,1\,7\,\mathit{xt}]\!]\,\phi\beta$. From the typing rules for Typ, we have $7 \in A'$. This is a counterexample showing $a \notin A'$ in (5.41) is not necessarily true. Fortunately, a correct rewrite results if we replace *remap* with a slightly different transformer in this case. The full rewrite is thus

$$\begin{aligned}
&\mathcal{R}\langle\!\langle \xi[\![\mathrm{Typ}\ c\,a\,(E : A'), =, \mathit{tgt}, \mathit{tgt\text{-}base}, \mathit{val\text{-}base}, \mathit{size}, \mathit{i\text{-}vars}]\!]\rangle\!\rangle\\
&\quad = a \in A' \rightarrow\\
&\qquad \langle\!\langle \xi[\![E, =, \mathit{tgt}, \mathit{tgt\text{-}base}, (\mathit{augmt}\ \mathit{val\text{-}base}\ a\ c), (\mathit{augmt}\ \mathit{size}\ a\ c), (\mathit{augmt}\ \mathit{i\text{-}vars}\ a\ c)]\!]\rangle\!\rangle,\\
&\qquad \langle\!\langle \xi[\![E, =, \mathit{tgt}, \mathit{tgt\text{-}base}, (\mathit{remap}\ \mathit{val\text{-}base}\ a\ c), (\mathit{remap}\ \mathit{size}\ a\ c), (\mathit{remap}\ \mathit{i\text{-}vars}\ a\ c)]\!]\rangle\!\rangle
\end{aligned}$$

where $(\mathit{augmt}\ f\ x\ y) = f\,[(f\ x)/y]$—*augmt* is like *remap* except it does not remove any element from $\mathrm{dom}\, f$.

Extr. Similar derivations yield rewrites for Extr.

$$\begin{aligned}
&\mathcal{R}\langle\!\langle \xi[\![\mathrm{Extr}^0_0\ c\,E, =, \mathit{tgt}, \mathit{tgt\text{-}base}, \mathit{val\text{-}base}, \mathit{size}, \mathit{i\text{-}vars}]\!]\rangle\!\rangle\\
&\quad = \langle\!\langle \xi[\![E, =, \mathit{tgt}, \mathit{tgt\text{-}base}, \mathit{val\text{-}base}\,[0/c], \mathit{size}, \mathit{i\text{-}vars}]\!]\rangle\!\rangle
\end{aligned}$$

$$\begin{aligned}
&\mathcal{R} = \langle\!\langle \xi[\![\mathrm{Extr}^0_1\ c\,E, =, \mathit{tgt}, \mathit{tgt\text{-}base}, \mathit{val\text{-}base}, \mathit{size}, \mathit{i\text{-}vars}]\!]\rangle\!\rangle\\
&\quad \langle\!\langle \xi[\![E, =, \mathit{tgt}, \mathit{tgt\text{-}base}, \mathit{val\text{-}base}\,[(\mathit{val\text{-}base}\ c){+}1/c], \mathit{size}, \mathit{i\text{-}vars}]\!]\rangle\!\rangle
\end{aligned}$$

Formal parameter references. Consider a code accumulation where the active assignment is for a reference to the ℓ-th formal function parameter.

$$\langle\!\langle \xi [\![x_\ell, =, tgt, tgt\text{-}base, val\text{-}base, size, i\text{-}vars]\!] \rangle\!\rangle \tag{5.42}$$

Suppose for the moment that $size = \emptyset$. The translation of the active site is then just

$$\begin{aligned} tgt[i_0 + (tgt\text{-}base\,0), \ldots, i_{m-1} + (tgt\text{-}base\,(m-1))] = \\ (\beta\, x_\ell)\ \mathbf{0}\ \mathbf{0}\,[(M\,c_1)/c_1] \cdots [(M\,c_q)/c_q] \end{aligned} \tag{5.43}$$

where M is specialized to

$$M\ c = \begin{cases} (i\text{-}vars\ c) + (val\text{-}base\ c) & \text{if } c \in \operatorname{dom} i\text{-}vars \\ (val\text{-}base\ c) & \text{otherwise} \end{cases} \tag{5.44}$$

By the assumptions of Property 5 on page 129, we can rewrite the right hand side of the equality (resp. assignment) in (5.43).

$$(\lambda j \,.\, M_i)\mathbf{0}\ \mathbf{0}\,[(M\ c_1)/c_1] \cdots [(M\ c_q)/c_q] = M_i\ \mathbf{0}\,[(M\ c_1)/c_1] \cdots [(M\ c_q)/c_q]$$

If we assume without loss of generality that $c_1 < c_2 < \cdots < c_q$, then this is equal to

$$a_\ell[(M\ c_1), \ldots, (M\ c_q)]$$

by Definition 5.5. We conclude that the proper rewrite is

$$\langle\!\langle \xi\ \ tgt[i_0 + (tgt\text{-}base\,0), \ldots, i_{q-1} + (tgt\text{-}base\,(q-1))] = a_\ell[(M\ c_1), \ldots, (M\ c_q)] \rangle\!\rangle$$

in the case where $size = \emptyset$.

When $size \neq \emptyset$, we can make progress by picking one element $\langle b_k, e_k \rangle$ in ran $size$ and emitting a $\forall$ loop for the target axis b_k. Here we need to consider the translation of the entire accumulation, not just the active site of interest. In particular, the full translation of (5.42) is if the form

$$\xi \underset{i_{b_1} \in 0:e_1}{\forall} \cdots \underset{i_{b_{k-1}} \in 0:e_{k-1}}{\forall} \underset{i_{b_k} \in 0:e_k}{\forall} \underset{i_{b_{k+1}} \in 0:e_{k+1}}{\forall} \cdots \underset{i_{b_m} \in 0:e_m}{\forall} tgt[\psi] = V\rho$$

where ρ, and ψ are fragments of the translation. This can be rewritten with the i_{b_k} loop outermost.

$$\xi \underset{i_{b_k}\in 0:e_k}{\forall} \left(\underset{i_{b_1}\in 0:e_1}{\forall} \cdots \underset{i_{b_{k-1}}\in 0:e_{k-1}}{\forall} \underset{i_{b_{k+1}}\in 0:e_{k+1}}{\forall} \cdots \underset{i_{b_m}\in 0:e_m}{\forall} tgt[\psi] = V\rho \right)$$

At this point we merely stop thinking of the i_{b_k} loop as part of the translation and emit it as code. The resulting rewrite is called *stripping* because it strips this $\forall$ from the ephemeral translation and makes it compiler output. It is

$$\langle\!\langle \xi \underset{i_{b_k}\in 0:e_k}{\forall} [\![x_i, =, \mathit{tgt}, \mathit{tgt\text{-}base}, \mathit{val\text{-}base}, (\mathit{rem}\; b_k\; \mathit{size}), (\mathit{add}\; i_{b_k}\; \mathit{size}\; \mathit{i\text{-}vars})]\!] \rangle\!\rangle$$

where $f = (\mathit{rem}\; x\; \mathit{size})$ is the same as *size* except $(f\; a)$ is undefined if $(\mathit{size}_1\; a) = x$, and $g = (\mathit{add}\; i_x\; \mathit{size}\; \mathit{i\text{-}vars})$ is the same as *i-vars* except for all s-val axes a such that $(\mathit{size}_1\; a) = x$, we have $(f\; a) = i_x$. Intuitively, *rem* removes mappings in *size* that take s-val axes to pairs containing x; *add* puts corresponding mappings to i_x in *i-vars* to record the newly emitted loop. This rewrite arises so often that it is worth stating as an auxiliary function. Assuming $\langle b_k, e_k \rangle \in \operatorname{ran} \mathit{size}$, we have

$$\begin{aligned} &\mathit{strip}(b_k, \langle\!\langle \xi [\![E, =, \mathit{tgt}, \mathit{tgt\text{-}base}, \mathit{val\text{-}base}, \mathit{size}, \mathit{i\text{-}vars}]\!] \rangle\!\rangle) = \\ &\qquad \langle\!\langle \xi \underset{i_{b_k}\in 0:e_k}{\forall} [\![E, =, \mathit{tgt}, \mathit{tgt\text{-}base}, \mathit{val\text{-}base}, (\mathit{rem}\; b_k\; \mathit{size}), (\mathit{add}\; i_{b_k}\; \mathit{size}\; \mathit{i\text{-}vars})]\!] \rangle\!\rangle \end{aligned}$$

Putting all this together, we have the complete rewrite for formal parameters. Let X be given by (5.42). Then

$$\begin{aligned} &\mathcal{R}\, X = (\mathit{size} = \emptyset) \rightarrow \\ &\qquad \langle\!\langle \xi\; \mathit{tgt}[i_0 + (\mathit{tgt\text{-}base}\, 0), \ldots, i_{q-1} + (\mathit{tgt\text{-}base}(q-1))] = a_\ell[(M\; c_1), \ldots, (M\; c_q)] \rangle\!\rangle \\ &\qquad (\mathit{strip}\; b_k\; X) \end{aligned}$$

where $\langle b_k, e_k \rangle \in \operatorname{ran} \mathit{size}$ and M is as in (5.44). The choice of b_k is arbitrary.[6]

[6]As a practical matter, one would typically choose b_k to provide e.g. closely spaced memory accesses. Though such second-order performance matters are important and interesting, they are beyond the scope of present discussion.

Reductions. The accumulation

$$\langle\!\langle \xi[\![\text{Red+ } c\,(E:\gamma), =, \mathit{tgt}, \mathit{tgt\text{-}base}, \mathit{val\text{-}base}, \mathit{size}, \mathit{i\text{-}vars}]\!]\rangle\!\rangle \tag{5.45}$$

submits to an analysis similar to that for formal parameters. If $\mathit{size} = \emptyset$, then the translation of the active site is

$$\begin{aligned} &\mathit{tgt}[i_0 + (\mathit{tgt\text{-}base}\,0), \ldots, i_{m-1} + (\mathit{tgt\text{-}base}\,(m-1))] \\ &\quad = \left(\lambda ij\,.\sum_{k=0}^{(\gamma\,c)-1} V\,i\,j\,[k/c]\right)\mathbf{0}\,[(M\ a_1)/a_1]\cdots\ [(M\ a_p)/a_p]\mathbf{0} \\ &\quad = \sum_{k=0}^{(\gamma\,c)-1} V\mathbf{0}\,[(M\ a_1)/a_1]\cdots\ [(M\ a_p)/a_p]\mathbf{0}\,[k/c] \end{aligned}$$

At this point it is clear that the rewrite can only be an assignment with a $\sum$ loop on the right side; but what about the loop body? Equation (5.35) shows that an active expression is appropriate. The desired rewrite is

$$\begin{aligned} &\langle\!\langle \xi\ \mathit{tgt}[i_0 + (\mathit{tgt\text{-}base}\,0), \ldots, i_{m-1} + (\mathit{tgt\text{-}base}\,(m-1))] = \\ &\qquad\qquad \sum_{k=0:(\gamma\,c)} [\![E, ?, \mathit{val\text{-}base}\,[0/c], \mathit{i\text{-}vars}\,[k/c]]\!]\rangle\!\rangle \end{aligned}$$

When size is non-empty, we merely strip. Let X be the accumulation in (5.45). Then the full rewrite equation is

$$\begin{aligned} &\mathcal{R}\,X = (\mathit{size} = \emptyset) \to \\ &\quad \langle\!\langle \xi\ \mathit{tgt}[i_0 + (\mathit{tgt\text{-}base}\,0), \ldots, i_{m-1} + (\mathit{tgt\text{-}base}\,(m-1))] = \\ &\qquad\qquad \sum_{k=0:(\gamma\,c)} [\![E, ?, \mathit{val\text{-}base}\,[0/c], \mathit{i\text{-}vars}\,[k/c]]\!]\rangle\!\rangle, \\ &\quad (\mathit{strip}\ b_k\ X) \end{aligned}$$

where $b_k \in (\text{ran } \mathit{size}_1)$.

Conditionals. Consider next the accumulation

$$\langle\!\langle \xi[\![\text{if } (B:A) \text{ then } T \text{ else } F, =, \mathit{tgt}, \mathit{tgt\text{-}base}, \mathit{val\text{-}base}, \mathit{size}, \mathit{i\text{-}vars}]\!]\rangle\!\rangle \tag{5.46}$$

Note A is the outer axis set of the predicate value. This set is significant because, intuitively speaking, loops whose variables index the array value of the predicate must also enclose the code for both branches—it would not be sensible for a loop to churn through elements of the predicate array value without evaluating the branches. Since the inner axis set of the predicate is empty, the only such loops are those corresponding to outer axes. Thus the strategy for rewriting is to strip target axes in the image of $A \cap (\mathrm{dom}\, size)$ under $size_1$, because we have no choice. Then, as usual, we look for an easily computed transformation of (5.46) for the base case, when $A \cap (\mathrm{dom}\, size)$ is empty.

Beginning with this base case, suppose $A \cap (\mathrm{dom}\, size) = \emptyset$ and let V_B, V_T, and V_F be the semantic values of B, T, and F in (5.46) with respect to ϕ and β as in Property 5, page 129. Then the translation of the active site is

$$\begin{aligned}
&\underset{i_{b_1}\in 0:e_1}{\forall} \cdots \underset{i_{b_m}\in 0:e_m}{\forall}\ tgt[i_0 + (tgt\text{-}base\,0), \ldots, i_{n-1} + (tgt\text{-}base\,(n-1))] \\
&\quad = (\lambda ij\,.\,(V_B\, i\, j) \rightarrow (V_T\, i\, j),\ (V_F\, i\, j)) \\
&\qquad\quad \mathbf{0}\,[(M\, a_1)/a_1] \cdots [(M\, a_p)/a_p]\ \mathbf{0}\,[(M\, c_1)/c_1] \cdots [(M\, c_q)/c_q] \\
&\quad = V_B\, \mathbf{0}\,[(val\text{-}base\, a_1) + (i\text{-}vars\, a_1)/a_1] \cdots [(val\text{-}base\, a_p) + (i\text{-}vars\, a_p)/a_p]\ \mathbf{0} \rightarrow \\
&\qquad\quad V_T\, \mathbf{0}\,[(M\, a_1)/a_1] \cdots [(M\, a_p)/a_p]\ \mathbf{0}\,[(M\, c_1)/c_1] \cdots [(M\, c_q)/c_q], \\
&\qquad\quad V_F\, \mathbf{0}\,[(M\, a_1)/a_1] \cdots [(M\, a_p)/a_p]\ \mathbf{0}\,[(M\, c_1)/c_1] \cdots [(M\, c_q)/c_q]
\end{aligned}$$

The intuitive reasoning above is thus verified. The value of the predicate of the conditional $\rightarrow$ does not depend on any $\forall$ loop of the interpretation. We can move these loops into the branches of the conditional. In the process, the functional $\rightarrow$ becomes a **proc** conditional assignment, but the meaning is unchanged.

$$\begin{aligned}
&\text{if } V_B\, \mathbf{0}\,[(val\text{-}base\, a_1) + (i\text{-}vars\, a_1)/a_1] \cdots [(val\text{-}base\, a_p) + (i\text{-}vars\, a_p)/a_p]\ \mathbf{0} \text{ then} \\
&\qquad \underset{i_{b_1}\in 0:e_1}{\forall} \cdots \underset{i_{b_m}\in 0:e_m}{\forall}\ tgt[i_0 + (tgt\text{-}base\,0), \ldots, i_{n-1} + (tgt\text{-}base\,(n-1))] = \\
&\qquad\qquad V_T\, \mathbf{0}\,[(M\, a_1)/a_1] \cdots [(M\, a_p)/a_p]\ \mathbf{0}\,[(M\, c_1)/c_1] \cdots [(M\, c_q)/c_q] \\
&\text{else} \\
&\qquad \forall_{i_{b_1}\in 0:e_1} \cdots \forall_{i_{b_m}\in 0:e_m}\ tgt[i_0 + (tgt\text{-}base\,0), \ldots, i_{n-1} + (tgt\text{-}base\,(n-1))] = \\
&\qquad\qquad V_F\, \mathbf{0}\,[(M\, a_1)/a_1] \cdots [(M\, a_p)/a_p]\ \mathbf{0}\,[(M\, c_1)/c_1] \cdots [(M\, c_q)/c_q]
\end{aligned}$$

The final step is to employ (5.35) on the predicate value and (5.31) on each of the branches to obtain the rewrite we might have predicted.

$$\begin{array}{l} \text{if } [\![B, ?, \mathit{val\text{-}base}, \mathit{i\text{-}vars}]\!] \\ \qquad \text{then } [\![T, =, \mathit{tgt}, \mathit{tgt\text{-}base}, \mathit{val\text{-}base}, \mathit{size}, \mathit{i\text{-}vars}]\!] \\ \qquad \text{else } [\![F, =, \mathit{tgt}, \mathit{tgt\text{-}base}, \mathit{val\text{-}base}, \mathit{size}, \mathit{i\text{-}vars}]\!] \end{array}$$

When $A \cap (\mathrm{dom}\, \mathit{size}) \neq \emptyset$, we strip to make progress. Let X be the accumulation in (5.46). Then the complete rewrite is

$$\begin{array}{l} \mathcal{R}\, X = (A \cap (\mathrm{dom}\, \mathit{size}) = \emptyset) \rightarrow \\ \qquad \langle\!\langle \xi \text{ if } [\![B, ?, \mathit{val\text{-}base}, \mathit{i\text{-}vars}]\!] \\ \qquad\quad \text{then } [\![T, =, \mathit{tgt}, \mathit{tgt\text{-}base}, \mathit{val\text{-}base}, \mathit{size}, \mathit{i\text{-}vars}]\!] \\ \qquad\quad \text{else } [\![F, =, \mathit{tgt}, \mathit{tgt\text{-}base}, \mathit{val\text{-}base}, \mathit{size}, \mathit{i\text{-}vars}]\!] \rangle\!\rangle, \\ \qquad \mathit{strip}(b, X) \end{array}$$

where $b = (\mathit{size}_1\, a)$ for some $a \in A \cap (\mathrm{dom}\, \mathit{size})$.

Join. We next turn to Join accumulations:

$$\langle\!\langle \xi [\![\text{Join}\, c\, (E_1 : C, \gamma)\, E_2, =, \mathit{tgt}, \mathit{tgt\text{-}base}, \mathit{val\text{-}base}, \mathit{size}, \mathit{i\text{-}vars}]\!] \rangle\!\rangle \tag{5.47}$$

Here no assumptions are necessary on axis sets; no stripping is performed by the rewrite function. However, rewrite cases arise in a different fashion. Consider the interpretation of the active site above:

$$\begin{array}{l} \underset{i_{b_1} \in 0:e_1}{\forall} \cdots \underset{i_{b_m} \in 0:e_m}{\forall} \mathit{tgt}[i_0 + (\mathit{tgt\text{-}base}\, 0), \ldots, i_{n-1} + (\mathit{tgt\text{-}base}\, (n-1))] \\ = \big(\lambda ij \,.\, (j\ c) < (\gamma_1'\ c) \ \rightarrow\ (V_1\, i\, j),\ (V_2\, i\, j\, [(j\ c) - (\gamma'\ c)/c])\big) \\ \qquad \mathbf{0}\, [(M\ a_1)/a_1] \cdots [(M\ a_p)/a_p]\ \mathbf{0}\, [(M\ c_1)/c_1] \cdots [(M\ c_q)/c_q] \\ = (M\ c) < (\gamma'\ c) \rightarrow \\ \qquad V_1\, \mathbf{0}\, [(M\ a_1)/a_1] \cdots [(M\ a_p)/a_p]\ \mathbf{0}\, [(M\ c_1)/c_1] \cdots [(M\ c)/c] \cdots [(M\ c_q)/c_q], \\ \qquad V_2\, \mathbf{0}\, [(M\ a_1)/a_1] \cdots [(M\ a_p)/a_p]\ \mathbf{0}\, [(M\ c_1)/c_1] \cdots [(M\ c)/c] \cdots [(M\ c_q)/c_q] \end{array} \tag{5.48}$$

where

$$\gamma' = \begin{cases} \gamma & \text{if} \quad c \in C \\ \gamma\, [1/c] & \text{otherwise} \end{cases} \tag{5.49}$$

to account for promotion of the value of E_1. The appearance of $(M\ c)$ in the conditional indicates that the nature of the rewrite depends on which of the three cases of the definition of M in (5.33) apply to c. We repeat the definition here with the cases lettered for reference.

$$M\ a = \begin{cases} i_b + (\textit{val-base}\, a) & \text{if } (\textit{size}_1\, a) = b & (a) \\ (\textit{i-vars}\, a) + (\textit{val-base}\, a) & \text{if } a \in (\text{dom } \textit{i-vars}) & (b) \\ (\textit{val-base}\, a) & \text{otherwise} & (c) \end{cases} \tag{5.50}$$

When c is in dom $\textit{size}_1$, then it satisfies (a) and $(M\ c) = i_b + (\textit{val-base}\, c)$ where $b = (\textit{size}_1\, c)$. This implies that the i_b loop is in the translation of the active site rather than in ξ, the code already emitted, and the condition $(M\ c) < (\gamma'\ c)$ in (5.48) is satisfied for the loop iterations where $i_b + (\textit{val-base}\, c) < (\gamma'\ c)$ or equivalently (recalling from Chapter 4 that $a \dot{-} b = a - b$ if $a \geq b$ and zero otherwise) when $i_b < (\gamma'\ c) \dot{-} (\textit{val-base}\ c)$.The condition is false in the other iterations. This allows a rewrite by "splitting" the i_b loop of the translation into two $\forall$ loops connected with $\wedge$ and having disjoint ranges covering the range of the original. Let $s = (\gamma'\ c) \dot{-} (\textit{val-base}\ c)$. Then the desired subranges are 0:min $\{(\textit{size}_2 c), s\}$ and s:$(\textit{size}_2 c) \dot{-} s$.

Suppose $X = \{x_1, \ldots, x_n\}$ and define $f\,[y/x]_{x \in X} = f\,[y/x_1] \cdots [y/x_n]$. Then the desired rewrite of (5.48) is

$$\begin{aligned} \langle\!\langle \xi \big(& [\![E_1, =, \textit{tgt}, \textit{tgt-base}, \textit{val-base}, \textit{size}\,[\langle b, \min\{s, (\textit{size}_2\, a)\}\rangle/a]_{a \in A}, \textit{i-vars}]\!] \wedge \\ & [\![E_2, =, \textit{tgt}, \textit{tgt-base}\,[(\textit{tgt-base}\, b) + s/b], \textit{val-base}\,[(\textit{val-base}\, a) \dot{-} s/a]_{a \in A}, \\ & \qquad\qquad \textit{size}\,[\langle b, (\textit{size}_2\, a) \dot{-} s\rangle/a]_{a \in A}, \textit{i-vars}]\!] \big) \rangle\!\rangle \end{aligned} \tag{5.51}$$

where $A = \{x \mid (\textit{size}_1\, x) = b\}$.

When $c \in (\text{dom } \textit{i-vars})$, then case (b) is satisfied; an i_b loop already exists in ξ. I.e.,

$$\xi = \rho \underset{i_b \in a:e}{\Lambda} \psi$$

where Λ is $\forall$ or Σ and $i_b = (\textit{i-vars}\, c)$. Referring again to (5.48), we see the condition $(M\ c) < (\gamma'\ c)$ is satisfied for just the i_b loop iterations where $i_b + (\textit{val-base}\ c) < (\gamma'\ c)$, that is when $i_b < (\gamma'\,c) \dot{-} (\mathit{val\text{-}base}\,c)$. Here we exploit the code accumulation to split the i_b loop, rewriting the context ξ as well as the accumulation. One way to do this is as follows. Let $s = (\gamma'\,c) \dot{-} (\mathit{val\text{-}base}\,c)$. Then we have

$$\begin{aligned}\langle\!\langle \rho\Big(&\bigwedge_{i_b \in q:s \dot{-} q} \psi[\![E_1, =, \mathit{tgt}, \mathit{tgt\text{-}base}, \mathit{val\text{-}base}, \mathit{size}, \mathit{i\text{-}vars}]\!] \parallel \\ &\bigwedge_{i_b \in s:e \dot{-} (s \dot{-} q)} \psi[\![E_2, =, \mathit{tgt}, \mathit{tgt\text{-}base}, \mathit{val\text{-}base}, \mathit{size}, \mathit{i\text{-}vars}]\!]\Big)\rangle\!\rangle\end{aligned} \tag{5.52}$$

where $\parallel$ is $\wedge$ if Λ is $\forall$ and $+$ if it is Σ. Another approach is to patch *tgt-base*, and *val-base* on the right of $\parallel$ and start the corresponding $\forall$ range at zero. But then we must also modify all references to i_b in the code of ψ.

This leaves case (c), where c is in neither $(\mathrm{dom}\,\mathit{size}_1)$ nor $(\mathrm{dom}\,\mathit{i\text{-}vars})$. Thus occurs for instance in compiling

$$\mathrm{Extr}_0^0\,0\,(\mathrm{Join}\,0\,v_1\,v_2)$$

Here the condition in (5.48) reduces to $(\mathit{val\text{-}base}\,c) < (\gamma'\,c)$. Since it contains no references to loop variables at all, we can change the conditional in the interpretation into a **proc** conditional that picks either the $(\mathit{val\text{-}base}\,c)$-th component of E_1 if $(\mathit{val\text{-}base}\,c) < (\gamma'\,c)$, or else the $(\mathit{val\text{-}base}\,c) - (\gamma'\,c)$-th component of E_2. Expressing this idea as a rewrite, we obtain

$$\begin{aligned}&\langle\!\langle \xi \text{ if } (\mathit{val\text{-}base}\,c) < (\gamma'\,c) \\ &\quad \text{then } [\![E_1, =, \mathit{tgt}, \mathit{tgt\text{-}base}, \mathit{val\text{-}base}, \mathit{size}, \mathit{i\text{-}vars}]\!] \\ &\quad \text{else } [\![E_2, =, \mathit{tgt}, \mathit{tgt\text{-}base}, \mathit{val\text{-}base}[(\mathit{val\text{-}base}\,c) - (\gamma'\,c)/c], \mathit{size}, \mathit{i\text{-}vars}]\!]\rangle\!\rangle\end{aligned} \tag{5.53}$$

Combining the three cases yields the full rewrite rule for Join assignments. Let X be the accumulation in (5.47). Then we have

$$\begin{aligned}\mathcal{R}X = {} & (\mathit{size}_1\,c = b) \rightarrow (5.51), \\ & (c \in \mathrm{dom}\,\mathit{i\text{-}vars}) \rightarrow (5.52), (5.53)\end{aligned}$$

User function calls. Perhaps the most interesting rewrite problem is the code accumulation for a function call.

$$\langle\!\langle \xi[\![(f\,(E : C')) : A, \alpha; C, \gamma,\ =, \mathit{tgt}, \mathit{tgt\text{-}base}, \mathit{val\text{-}base}, \mathit{size}, \mathit{i\text{-}vars}]\!]\rangle\!\rangle \tag{5.54}$$

The translation of the active site is

$$\begin{aligned}
&\underset{i_{b_1}\in 0:e_1}{\forall}\cdots\underset{i_{b_m}\in 0:e_m}{\forall} tgt[i_0+(\mathit{tgt\text{-}base}\,0),\ldots,i_{n-1}+(\mathit{tgt\text{-}base}\,(n-1))]\\
&\quad=(\lambda i\,.\,(\phi\, f)(\mathcal{E}[\![E]\!]\,\phi\beta\, i))\ \mathbf{0}\,[(M\,a_1)/a_1]\cdots[(M\,a_p)/a_p]\ \mathbf{0}\,[(M\,c_1)/c_1]\cdots[(M\,c_q)/c_q]\\
&\quad=(\phi\, f)\,(\mathcal{E}[\![E]\!]\,\phi\beta\ \mathbf{0}\,[(M\,a_1)/a_1]\cdots[(M\,a_p)/a_p])\ \mathbf{0}\,[(M\,c_1)/c_1]\cdots[(M\,c_q)/c_q]
\end{aligned}\tag{5.55}$$

where $a_1,\ldots,a_p \notin (\mathrm{dom}\,\mathit{size})$. Assuming without loss of generality that $c_1 < c_2 < \cdots < c_{|C|}$, we can rewrite this as

$$\begin{aligned}
&\underset{i_{b_1}\in 0:e_1}{\forall}\cdots\underset{i_{b_m}\in 0:e_m}{\forall} tgt[i_0+(\mathit{tgt\text{-}base}\,0),\ldots,i_{n-1}+(\mathit{tgt\text{-}base}\,(n-1))] =\\
&\qquad\qquad s[(M\,c_1),\ldots,(M\,c_{|C|})]
\end{aligned}\tag{5.56}$$

where the value of s is established by a procedure call $f(t;s)$. If a correct (by Definition 5.7) compilation of f is available, then we have that (5.56) assigns correct values as long as

$$t=\mathbf{T}\,\mathcal{E}[\![E]\!]\,\phi\beta\ \mathbf{0}\,[(M\,a_1)/a_1]\cdots[(M\,a_p)/a_p]$$

This equality is the translation of the code accumulation

$$[\![E,=,t,0_{0:|C'|},0'_C,\lambda c.\langle(q_{C'}\,c),(\gamma\, c)\rangle,\emptyset]\!]\tag{5.57}$$

as long as $(\mathrm{dom}\,\mathit{size})\cap A=\emptyset$. The derivation is similar to (5.34) for the function body accumulation (5.26). Combining these observations in a single rewrite yields

$$\begin{aligned}
&\langle\!\langle \xi\ \text{let } t\ \rhd\ [\![E,=,t,0_{0:|C'|},0_{C'},\lambda c.\langle(q_{C'}\,c),(\gamma\, c)\rangle,\emptyset]\!]\\
&\quad \text{in let } s\ \rhd\ f(t;s)\\
&\quad \text{in } \underset{i_{b_1}\in 0:e_1}{\forall}\cdots\underset{i_{b_m}\in 0:e_m}{\forall} tgt[i_0+(\mathit{tgt\text{-}base}\,0),\ldots,i_{n-1}+(\mathit{tgt\text{-}base}\,(n-1))] =\\
&\qquad\qquad s[(M\,c_1),\ldots,(M\,c_{|C|})]\rangle\!\rangle
\end{aligned}\tag{5.58}$$

If $(\mathrm{dom}\,\mathit{size})\cap A\neq\emptyset$, we strip to make progress. Let X be the accumulation in (5.54). Then the complete rewrite rule is

$$\mathcal{R}\,X=(\mathrm{dom}\,\mathit{size}\cap A=\emptyset)\rightarrow(5.58),\ (\mathit{strip}\,b_k\,X)\tag{5.59}$$

where $(\mathit{size}_1\,a)=b_k$ for any $a\in(\mathrm{dom}\,\mathit{size})\cap A$. This discussion generalizes easily to function calls with multiple arguments.

Optimizing procedure calls. While (5.59) is correct, it contains important opportunities for improvement. For a simple example, consider the **svexp** function

$$f\,x \equiv g\,x$$

where x is known to be a vector. For this we obtain

$$\begin{aligned} f(x;r) \;\triangleright\; & \text{let } t \;\triangleright \mathop{\forall}_{i\in 0:d} t[i] = x[i] \\ & \text{in let } s \;\triangleright g(t;s) \\ & \text{in } \mathop{\forall}_{i\in 0:d} r[i] = s[i] \end{aligned}$$

when an obviously correct and more efficient alternative is

$$f(x;r) \;\triangleright g(x;r)$$

The remainder of this section is concerned with transformations on **proc** code that make procedure calls more efficient.

Simple copy elimination. **proc** code of the form

$$\begin{aligned} & \text{let } t \;\triangleright \mathop{\forall}_{i_0\in 0:d_0(t)} \cdots \mathop{\forall}_{i_{n(t)-1}\in 0:d_{n(t)-1}} t[i_0,\ldots,i_{n(t)-1}] = x[i_{p_0},\ldots,i_{p_{n(t)-1}}] \\ & \text{in } B \end{aligned}$$

with $p_0,\ldots,p_{n(t)-1}$ a permutation of $0,\ldots,n(t)-1$ is called a *simple copy.* It may be replaced with the let body B, wherein all occurrences of t are replaced with an appropriately transposed x:

$$[p_0,\ldots,p_{n(t)-1}]x$$

This is immediate from (5.20) and (5.21).

Another form of simple copy occurs when the target of a procedure call is at least as big as the procedure call result. In this case there is no need for a let to allocate fresh storage for the result. Consider lets of the following form, which can result from (5.59):

$$\begin{array}{l} \text{let } s \rhd f(t; s) \\ \text{in } \underset{i_{b_0} \in 0:d_0(s)}{\forall} \cdots \underset{i_{b_{n(s)-1}} \in 0:d_{n(s)-1}}{\forall} tgt[I_0, \ldots, I_{n(tgt)-1}] = s[i_{b_0}, \ldots, i_{b_{n(s)-1}}] \end{array} \tag{5.60}$$

where I_k for $0 \leq k < n(tgt)$ are indexers and the order of $\forall$ loops may be arbitrarily permuted. Let $I_{q_0}, \ldots, I_{q_{n(s)-1}}$ be the subsequence of $I_0, \ldots, I_{n(tgt)-1}$ where I_{q_ℓ} is of the form $i_{b_k} + B_{q_\ell}$, that is, those indexers that include one of the loop variables i_{b_k}, $0 \leq k < n(s)$ and an arithmetic expression arising from *tgt-base* during compilation. The other indexers contain variables of ancestral loops or have no loop variables at all. As an optimization, we can replace (5.60) with

$$f(t; [p_0, \ldots, p_{n(tgt)-1}]\, tgt[I'_0, \ldots, I'_{n(tgt)-1}])$$

where

$$I'_j = \begin{cases} B_j{:}d_k(s) & \text{if } I_j = B_j + i_{b_k} \\ I_j & \text{otherwise} \end{cases}$$

and

$$p_j = \begin{cases} q_k & \text{if } j = q_\ell \text{ for some } \ell \text{ and } I_j = B_j + i_{b_k} \\ j & \text{otherwise} \end{cases}$$

That is, rather than allocate a fresh array for the procedure call result, we arrange for the call to deposit its result in the correct portion of the target.

5.5.2 Rewriting active expressions

The principles for deriving active expression rewrites is the same as for assignments, and the algebra is somewhat simpler. We present the results without elaboration.

Proj.

$$\begin{aligned} &\mathcal{R}\langle\!\langle \xi[\![\text{Proj } a\, c\, E, ?, \textit{val-base}, \textit{i-vars}]\!]\rangle\!\rangle \\ &\qquad = \langle\!\langle \xi[\![E, ?, (\textit{remap val-base } c\, a), (\textit{remap i-vars } c\, a)]\!]\rangle\!\rangle \end{aligned}$$

Typ.

$$\begin{aligned} &\mathcal{R}\langle\!\langle \xi[\![\text{Typ } c\, a\, (E : A'), ?, \textit{val-base}, \textit{i-vars}]\!]\rangle\!\rangle \\ &\qquad = a \in A' \rightarrow \langle\!\langle \xi[\![E, ?, (\textit{augmt val-base } a\, c), (\textit{augmt i-vars } a\, c)]\!]\rangle\!\rangle, \\ &\qquad\qquad \langle\!\langle \xi[\![E, ?, (\textit{remap val-base } a\, c), (\textit{remap i-vars } a\, c)]\!]\rangle\!\rangle \end{aligned}$$

Extr.

$$\mathcal{R}\langle\!\langle \xi [\![\mathrm{Extr}_0^0\; c\, E, ?, \mathit{val\text{-}base}, \mathit{i\text{-}vars}]\!] \rangle\!\rangle = \langle\!\langle \xi [\![E, ?, \mathit{val\text{-}base}\,[0/c], \mathit{i\text{-}vars}]\!] \rangle\!\rangle$$

$$\mathcal{R}\langle\!\langle \xi [\![\mathrm{Extr}_1^0\; c\, E, ?, \mathit{val\text{-}base}, \mathit{i\text{-}vars}]\!] \rangle\!\rangle = \langle\!\langle \xi [\![E, ?, \mathit{val\text{-}base}\,[(\mathit{val\text{-}base}\, c){+}1/c], \mathit{i\text{-}vars}]\!] \rangle\!\rangle$$

Formal parameter references. Let M be as in (5.44). Then we have

$$\mathcal{R}\,\langle\!\langle \xi [\![(x_\ell : C), ?, \mathit{val\text{-}base}, \mathit{i\text{-}vars}]\!] \rangle\!\rangle = \langle\!\langle \xi\; a_\ell[(M\; c_1), \ldots, (M\; c_q)] \rangle\!\rangle$$

where without loss of generality we assume $C = \{c_1 < c_2 < \cdots < c_q\}$.

Reductions.

$$\langle\!\langle \xi [\![\mathrm{Red}{+}\; c\,(E : \gamma), ?, \mathit{val\text{-}base}, \mathit{i\text{-}vars}]\!] \rangle\!\rangle = \langle\!\langle \xi \sum_{k=0:(\gamma\, c)} [\![E, ?, \mathit{val\text{-}base}\,[0/c], \mathit{i\text{-}vars}\,[k/c]]\!] \rangle\!\rangle$$

Conditionals.

$$\begin{aligned}\langle\!\langle \xi [\![\text{if } B \text{ then } T \text{ else } F, ?, \mathit{val\text{-}base}, \mathit{i\text{-}vars}]\!] \rangle\!\rangle = \langle\!\langle \xi \;& \text{if } [\![B, ?, \mathit{val\text{-}base}, \mathit{i\text{-}vars}]\!] \\ & \text{then } [\![T, ?, \mathit{val\text{-}base}, \mathit{i\text{-}vars}]\!] \\ & \text{else } [\![F, ?, \mathit{val\text{-}base}, \mathit{i\text{-}vars}]\!] \rangle\!\rangle\end{aligned}$$

Join.

$$\mathcal{R}\langle\!\langle \xi [\![\mathrm{Join}\; c\,(E_1 : C, \gamma)\, E_2, ?, \mathit{val\text{-}base}, \mathit{i\text{-}vars}]\!] \rangle\!\rangle = c \in (\mathrm{dom}\; \mathit{i\text{-}vars}) \rightarrow (5.61), (5.62)$$

The two cases (5.61) and (5.62) are as follows. First, we have $c \in (\mathrm{dom}\; \mathit{i\text{-}vars})$; thus

$$\xi = \rho \mathop{\Lambda}_{i_b = q:c} \psi$$

Let $s = (\gamma'\, c) \dot{-} (\mathit{val\text{-}base}\, c)$ with γ' as in (5.49). Then the rewrite is

$$\langle\!\langle \rho(\mathop{\Lambda}_{i_b \in q:s \dot{-} q} \psi[\![E_1, ?, \mathit{val\text{-}base}, \mathit{i\text{-}vars}]\!] \;||\; \mathop{\Lambda}_{i_b \in s:e \dot{-} (s \dot{-} q)} \psi[\![E_2, ?, \mathit{val\text{-}base}, \mathit{i\text{-}vars}]\!]) \rangle\!\rangle \tag{5.61}$$

For the other case, *val-base* selects a value from E_1 or E_2.

$$\begin{aligned}\langle\!\langle \xi \text{ if } (\mathit{val\text{-}base}\; c) < (\gamma'\, c) \text{ then } & [\![E_1, ?, \mathit{val\text{-}base}, \mathit{i\text{-}vars}]\!] \\ \text{else } & [\![E_2, ?, \mathit{val\text{-}base}\,[(\mathit{val\text{-}base}\, c) \dot{-} (\gamma'\, c)/c], \mathit{i\text{-}vars}]\!] \rangle\!\rangle\end{aligned} \tag{5.62}$$

User function calls.

$$\mathcal{R}\langle\!\langle \xi[\![(f\,(E : C')) : C, \gamma,\ ?, \textit{val-base}, \textit{i-vars}]\!]\rangle\!\rangle =$$
$$\begin{array}{l}\langle\!\langle \xi \text{let } t \ \rhd [\![E, =, t, 0_{0:|C'|}, 0'_C, \lambda c.\langle (q_{C'}\, c), (\gamma\, c)\rangle, \emptyset]\!] \\ \quad \text{in let } s \ \rhd f\!\!f(t; s) \\ \quad \text{in } s[(M\, c_1), \ldots, (M\, c_{|C|})]\end{array}$$

where again $C = \{c_1 < c_2 < \cdots < c_{|C|}\}$.

5.5.3 Loop nest optimizations.

Though the compiler as given emits code only when necessary, some of the loop nests it constructs are inefficient. One example is the following.

Theorem 5.8 *The size of* $\mathcal{C}\,\langle\!\langle \xi \rangle\!\rangle$ *as given is* $O(2^{|\xi|})$.

Proof. For convenience, assume

$$\text{Join}\, c\, E_1 \cdots E_n = \text{Join}\, c\, E_1\, (\text{Join}\, c\, E_2 (\cdots (\text{Join}\, c\, E_{n-1}\, E_n) \cdots))$$

Then consider the function

$$\begin{array}{rl} f\, v_1\, w_1 \cdots v_n\, w_n \equiv & \text{Proj}\, 7\, 1\ \text{Red} +\ 0\ \text{Typ}\, 1\, 7 \\ & \quad \text{Join}\, 0\, (\text{Join}\, 1\, v_1\, w_1) \\ & \qquad\qquad \vdots \\ & \qquad\quad (\text{Join}\, 1\, v_n\, w_n) \end{array}$$

It is straightforward to show that this function contains 2^n loops. $\square$

This behavior is not worrisome in that it appears only for pathological inputs. Moreover, it is straightforward to derive replacements for the loop splitting rewrites that cause the problem. We leave this to the reader; the essence of the revised rules is in (5.62) and also [Bud88]. Practically speaking, the rewritten rules generate somewhat slower code than splitting, so a compiler implementation would do well to split loops unless the resulting code would be too big.

A more important problem is that invariant calculations may be repeated needlessly within loops. For this, we apply the principle of the classical optimization *loop invariant removal.*

Example 5.9 Consider the following function

$$\begin{array}{l} \mathit{foo}\, x\, y \equiv \mathrm{Proj}\, 7\, 0 \\ \qquad\qquad \mathrm{Red}+\ 0\, (\mathrm{Join}\, 0\, \mathrm{Red}+\ 0\, x \\ \qquad\qquad\qquad\qquad\qquad \mathrm{Typ}\, 0\, 7\, y) \end{array}$$

which compiles to[7]

$$\mathit{foo}(x, y; r) \ \rhd \bigforall_{i_0 \in 0:d_0(r)} r[i_0] = y[i_0] + \sum_{k \in 0:d_0(x)} x[k]$$

We need compute the summation only once:

$$\begin{array}{ll} \mathit{foo}(x, y; r) \ \rhd & \text{let } t \ \rhd t[] = \displaystyle\sum_{k \in 0:d_0(x)} x[k] \\ & \text{in} \displaystyle\bigforall_{i_0 \in 0:d_0(r)} r[i_0] = y[i_0] + t \end{array}$$

□

In general, if we have a **proc** $\langle \mathit{assign} \rangle$ term of the form

$$\xi \bigwedge_i \rho \quad \text{where } \rho \text{ contains } \sum_k E$$

If E does not contain references to i, then we can write this as

$$\xi \text{ let } t \ \rhd t[\,] = E \text{ in } \rho'$$

where ρ' is ρ with instances of $\sum_k E$ replaced by t.

We need not defer this optimization until after compilation. We can modify the rewrite rule for Red+ to inspect and transform its input code accumulation as desired.

Generalizing once more, we can modify all rewrite rules that potentially generate code for expensive operations that can be moved farther out in the loop nest. Another example with clear profit is to transform procedure call code

$$\xi \bigforall_k \rho \text{ let } t \ \rhd E_1 \text{ in let } s \ \rhd f(t; s) \text{ in } E_2$$

[7] We have applied some algebraic transformations, i.e. eliminating loops that execute exactly once.

If i does not appear in E_1, and E_2 is an assignment we can rewrite this as

$$\xi\ \rho\ \text{let}\ t\ \rhd E_1\ \text{in let}\ s\ \rhd f(t;s)\ \text{in}\ \bigvee_k E_2$$

Other examples include choosing evaluation order of procedure call arguments to obtain efficient nestings and moving the evaluation of conditional predicate code outward. The latter may involve changing a conditional expression to a conditional assignment.

5.6 Perspective

Let us take stock of the discussion thus far. Our compiler translates the high level manipulations of typical component spaces in **svexp** to a single assignment language, **proc**. The techniques given here—code accumulations and algebraic compiler development using translations—are fundamental for s-vals in the same sense that e.g. closures and various virtual machine models are fundamental to compiling general functions. Thus the result of this chapter is not the compiler itself, but the methodology that led to it, which should be useful for any language based on s-vals.

Chapter 6

Evaluation In Place

Previous chapters described the functional matrix language **svexp** and gave a compiler that accepts type-annotated **svexp** programs and returns compilations to a single assignment array language called **proc**. In turn, **proc** programs are easy to transform to machine codes for a variety of architectures. They are memory efficient in certain respects. However, many compiled **proc** programs still take asymptotically more storage (and copying time) than required for the problems they solve. This inefficiency and an optimization algorithm to eliminate it are the topics of this chapter.

6.1 Overview

We first consider an instance where the compiler of Chapter 5 yields code as efficient as an imperative language compiler. This success stems from two desirable properties that careful design of the compiler has provided:

- It allocates storage only for a single inner matrix of each user function call argument and result.

- It detects opportunities to eliminate these allocations when the storage serves as an unnecessary copy of a portion of the caller's argument or of the result value.

For example, these provide good results with the following function to reverse a vector:

$$\begin{aligned} rev\ x \equiv\ & \text{if } (\text{NumOf } x\ 0\ 1) \text{ then } x \\ & \text{else Join } 0\ (rev\ \text{Extr}_1^0\ 0\ x)\ (\text{Extr}_0^0\ 0\ x) \end{aligned}$$

This compiles to the following **proc** code. Assume $x : n$.

$$\begin{aligned} rev(x;\ r)\ \rhd\ & \text{if } n = 1 \text{ then } \underset{i \in 0:n}{\forall}\ r[i] = x[i] \\ & \text{else } rev(x[1\!:n{-}1];\ r[0\!:n{-}1]) \wedge r[n{-}1] = x[0] \end{aligned}$$

Recall that we can pass arguments by reference (that is, as pointers) in **proc** due to its single assignment property. Further, the recursive side of $\wedge$ is tail recursive if we evaluate it last. When tail recursion is removed and $n = 1$ propagated through the true branch of the if, what remains is an efficient and concise code for reversing x into r. Using a C representation for arrays as in Figure 5.2 and Table 5.1, we obtain

```
void rev(x, xs, d1, r, rs)
{  start:
   if (d1 == 1)
      r[0] = x[0];
   else {
      r[rs * (d1 - 1)] = x[0];
      x = x + xs; d1 = d1 - 1;
      goto start;
} }
```

This example is noteworthy because comparable code to reverse *lists* is well-studied, owing to its notorious inefficiency. For instance, this Common Lisp code requires $O(n^2)$ time and space.

```
(defun rev (x)
  (if (null (cdr x))
    x
    (append (rev (cdr x))
            (list (car x)))))
```

Replacing *append* with the destructive equivalent *nconc* gets the space requirement down to $O(n)$, but time is still $O(n^2)$. Moreover, the programmer must then remember that the original (unreversed) value of x is destroyed by the call ($rev\ x$).

6.1.1 The remaining problem

Unfortunately, the compiler as it stands is still not good enough. Many **svexp** functions compile to code that is asymptotically less efficient in space and run time than natural imperative code for the same purpose. The problem is rooted in the single assignment property of **proc** that until now has made things simpler.

Example 6.1 Consider the following function.

$$\begin{array}{l} g\,x \equiv \text{if NumOf}\,x\,0\,0 \text{ then } x \\ \qquad \text{else Join}\,0\,(\text{Extr}_0^0\,0\,x) \\ \qquad\qquad \text{Proj}\,7\,0\,(+\,1 \\ \qquad\qquad\qquad \text{Typ}\,0\,7\,(g\,\text{Proj}\,7\,0\,(+\,1 \\ \qquad\qquad\qquad\qquad \text{Typ}\,0\,7(\text{Extr}_1^0\,0\,x))) \end{array}$$

This accepts an inner n-vector x and laboriously adds $2k$ to element k for $0 \leq k < n$. With the compiler and optimizations of Chapter 5 we obtain

$$\begin{array}{l} g(x;r) \ \rhd \text{if } n = 0 \text{ then } \underset{i \in 0:n}{\forall} r[i] = x[i] \\ \qquad \text{else } r[0] = x[0] \wedge \text{let } t \ \rhd \underset{i \in 0:n-1}{\forall} t[i] = 1 + x[i+1] \\ \qquad\qquad \text{in let } s \ \rhd g(t;s) \\ \qquad\qquad \text{in } \underset{i \in 0:n-1}{\forall} r[i+1] = 1 + s[i] \end{array}$$

□

What has gone wrong? There are no simple copies in this code. The compilation executes $n-1$ recursive calls, each with a computation into $O(n)$ fresh storage. Thus the total cost of storage is $O(n^2)$. The simple copy elimination of Chapter 5 was a step in the right direction, but we must replace it with something more powerful.

6.1.2 A solution

Example 6.1 shows that the single assignment property of **proc** is an obstacle to efficiency that must be overcome. The crux of the problem is that the asymptotic storage performance of a **proc** program with no reductions and where simple copy elimination does not apply cannot be better than the time performance of the underlying algorithm. For instance, a $O(n^2)$ sort algorithm coded in **proc** needs $O(n^2)$ storage. The example above shares this malady.

On the other hand, if multiple assignments to the same array element were allowed, we could rewrite Example 6.1, hence many other procedures, to compute the same function as before rewriting, but using less storage, $O(n)$ in the example. This observation is the basis for the approach presented here for improving storage efficiency of **proc** programs. We proceed in outline as follows:

- We lift the single assignment restriction of **proc** by extending **proc** to a very similar language, $\mathbf{proc_2}$, which allows two or more subarrays to share the same storage. An explicit *assignment order* is necessary to make $\mathbf{proc_2}$ useful.

- Hence the goal is to transform a **proc** procedure f to a $\mathbf{proc_2}$ procedure f' that is *extensionally equivalent* to f (we say $f \equiv f'$). That is, the call $f(a_1, \ldots, a_n; s)$ terminates if and only if $f'(a_1, \ldots, a_n; s')$ terminates, and $s = s'$.

- The assignment order for $\mathbf{proc_2}$ admits a succinct *sharing condition* that describes when two arrays in f can share storage without changing the result computed by f.

- A test of the body syntax of f conservatively indicates if the sharing condition will certainly be met for a given array variable a and subarray b. Hence we can decide in many useful cases at compile time that a and b can share storage.

- The sharing condition also leads to an algorithm to deduce a small set of pairs $[b/a]$ as above that are likely *prospects* for storage sharing. On finding a pair that passes the syntactic test for the sharing condition, f in **proc** or $\mathbf{proc_2}$ can be transformed to $f' \in \mathbf{proc_2}$, $f' \equiv f$, where f' has storage performance generally better than f.

6.2 $\mathbf{proc_2}$ sharing syntax

We define $\mathbf{proc_2}$ procedures to be transformations from **proc** procedures:

$$\mathbf{proc_2} = \bigcup_{\substack{k \geq 0 \\ f \in \mathbf{proc}}} (share^k\, f)$$

where, roughly speaking, $(share\, f)$ is the set of procedures where two variables in f share storage.

More precisely, consider all pairs $[b/a]$ where a is an array variable in f and b is a subarray in f of the same dimension as a. Then $share(f)$ contains all programs f' constructed from f by replacing a with b and f with f' throughout f. Further, if a happens to be a *let*–defined variable[1] then replace

$$\text{let } a \;\rhd \xi \qquad \text{with simply} \qquad \text{let } \xi$$

This shows that the *let* no longer allocates a fresh array, but redefines elements of b instead. We call this a *sharing transformation* and write it

$$f' = f\,[b/a]$$

Example 6.1 provides a useful illustration of sharing transformations. First, let $g' = g\,[x/r]$ to obtain the $\mathbf{proc_2}$ program

[1]We assume without loss of generality that *let*-defined variables are distinct.

$$
\begin{aligned}
g'(x;x) \;\rhd\; & \text{if } n = 0 \text{ then } \underset{i\in 0:n}{\forall} x[i] = x[i] \\
& \text{else } x[0] = x[0] \wedge \text{let } t \;\rhd \underset{i\in 0:n-1}{\forall} t[i] = 1 + x[i+1] \\
& \qquad \text{in let } s \;\rhd\; g'(t;s) \\
& \qquad \text{in } \underset{i\in 0:n-1}{\forall} x[i+1] = 1 + s[i]
\end{aligned}
\tag{6.1}
$$

When a procedure's return array shares storage with a formal parameter like this, we say the procedure operates *in place.* When e.g. g' operates in place, a procedure call $g'(t;s)$ where $s \neq t$ writes its result in array t and implicitly copies it to s. This behavior is also consistent with sharing of the return array with a proper subarray of an parameter, e.g. if procedure p has a parameter $x : m \times n$ and return array $r : n$, sharing such as $p\,[x[0, 0{:}n]/r]$ is appropriate, and the call $p(t;s)$ computes the result into $t[0, 0{:}n]$, then copies this subarray to s.[2]

For code generated by the **svexp** compiler, calls to in–place procedures are always opportunities to share the call's result array with the respective argument subarray by transformations. In this case, let $g'' = g'\,[t/s]$ to obtain

$$
\begin{aligned}
g''(x;x) \;\rhd\; & \text{if } n = 0 \text{ then } \underset{i\in 0:n}{\forall} x[i] = x[i] \\
& \text{else } x[0] = x[0] \wedge \text{let } t \;\rhd \underset{i\in 0:n-1}{\forall} t[i] = 1 + x[i+1] \\
& \qquad \text{in let } g''(t;t) \\
& \qquad \text{in } \underset{i\in 0:n-1}{\forall} x[i+1] = 1 + t[i]
\end{aligned}
\tag{6.2}
$$

For calls of the hypothetical p above, $p(t;s)$, the transformation is $\cdot\,[t[0, 0{:}n]/s]$. With these, the implicit copies described above are avoided.[3] Finally, let $g''' = g''\,[x[1{:}n-1]/t]$ so we have

[2]However, consider the inverse situation: procedure q has an n-vector parameter and an $m \times n$ result. Then it is desirable to share a parameter with a portion of the return array, i.e. $q' = q\,[r[0, 0{:}n]/x]$. A call $q'(t;s)$ implicitly copies the argument t to $s[0, 0{:}n]$, then computes the result in s.

[3]However, consider the similar transformation $\cdot\,[s[0, 0{:}n]/t]$ applied to any procedure calling q' as in the previous footnote. Since such calls are invariably of the form let $t \;\rhd\; \xi$ let $s \;\rhd\; q'(t;s)$ in$\cdots$, we find the sharing transformations defined

$$
\begin{aligned}
g'''(x;x) \;\rhd\; & \text{if } n=0 \text{ then } \underset{i\in 0:n}{\forall}\, x[i]=x[i] \\
& \quad \text{else } x[0]=x[0] \wedge \text{let } \underset{i\in 0:n-1}{\forall}\, \underline{x[i+1]} = 1 + x[i+1] \\
& \qquad\qquad\qquad\qquad \text{in let } g'''(x[1{:}n-1]; x[1{:}n-1]) \\
& \qquad\qquad\qquad\qquad \text{in } \underset{i\in 0:n-1}{\forall}\, x[i+1] = 1 + \underline{x[i+1]}
\end{aligned} \tag{6.3}
$$

The underlined expressions are simplifications of $x[1{:}n-1][i]$. Simplifying further by the elimination of "dead code", we obtain

$$
\begin{aligned}
g'''(x;x) \;\rhd\; & \text{if } n=0 \text{ then } \{\textit{do nothing}\} \\
& \quad \text{else let } \underset{i\in 0:n-1}{\forall}\, x[i+1] = 1 + x[i+1] \\
& \qquad \text{in let } g'''(x[1:n-1]; x[1:n-1]) \\
& \qquad \text{in } \underset{i\in 0:n-1}{\forall}\, x[i+1] = 1 + x[i+1]
\end{aligned} \tag{6.4}
$$

What do these transformations achieve? In the interpretation of $\mathbf{proc}_2$ given below, this procedure computes the same result as the original, but the answer appears in the storage originally provided for the argument. Rather than the $O(n^2)$ storage needed by the original procedure, this one uses only $O(n)$. We contrived this example to show that the strong link between run time and storage requirement for **proc** discussed in Section 6.1.2 has been broken. Indeed, the transformed program still has the $O(n^2)$ run time of the original. However, there exist many other codes whose run times are improved by sharing transformations because sharing reduces overhead due to copying.

---

here to be inadequate. In the example case we obtain let $\xi\,[s[0,0{:}n]/t]$ in let $s \;\rhd\; q'(s[0,0{:}n];s)$ in $\cdots$. Thus the outer *let* assigns to array s outside the *let* that defines s. For a remedy we could extend sharing transformations to shift the allocation of s to the outer *let* as necessary, i.e. let $s \;\rhd\; \xi\,[s[0,0{:}n]/t]$ in let $q'(s[0,0{:}n];s)$ in $\cdots$ This kind of sharing is important and useful in practice. For simplicity, however, we refrain from carrying the extended transformation through succeeding discussions.

6.3 Assignment ordering

The interpretation of $\mathbf{proc_2}$ programs is identical to **proc** in both declarative and imperative senses, except for one serious complication. The single assignment property of **proc** leaves no doubt as to the value of each array element. We need only identify the single assignment that defines the element's value—an "=" construct, procedure call, or the implicit definition of formal parameters. This imposes a partial *evaluation order* on **proc** terms—an array element must be assigned before its value is used; a conditional predicate's value must be known before the value established by the branches. Such implicit orderings are conventionally called *data flow* constraints. Conversely, as e.g. (6.4) shows, $\mathbf{proc_2}$ allows multiple assignments to the same array element. We must determine which assigned value is "seen" by each reference.

Example 6.2 Consider the **proc** procedure

$$\begin{array}{rl} h(x;r) \;\rhd & \text{let } t \;\rhd \displaystyle\mathop{\forall}_{i\in 0:d_0(x)} t[i] = x[i]+1 \\ & \quad \text{in } \displaystyle\mathop{\forall}_{i\in 0:d_0(x)} r[i] = t[i] \end{array}$$

An evaluation order for array elements satisfying data flow constraints, (after the implicit initial assignment of values to x when h is called) is

$$t[0] = x[0]+1,\ r[0] = t[0],\ t[1] = x[1]+1,\ r[1] = t[1],\ \ldots$$

simply because the value of each right hand side of = is defined. Any data flow order leads h to return the same answer because every use of an array element value corresponds to a single assignment.

However, consider $h' = h\,[t/x]$, i.e.

$$\begin{array}{rl} h'(x;r) \;\rhd & \text{let } \displaystyle\mathop{\forall}_{i\in 0:d_0(x)} x[i] = x[i]+1 \\ & \quad \text{in } \displaystyle\mathop{\forall}_{i\in 0:d_0(x)} r[i] = x[i] \end{array}$$

Here the assignments corresponding to the above are

$$x[0] = x[0] + 1,\ r[0] = x[0],\ x[1] = x[1] + 1,\ r[1] = x[1],\ \ldots$$

and this order yields the same result as the original procedure. However, dataflow alone permits other orders that do not yield the same result. For instance the alternative order

$$r[0] = x[0],\ x[0] = x[0] + 1,\ r[1] = x[1],\ x[1] = x[1] + 1,\ \ldots$$

has all right hand sides defined, but yields a different result array r. □

Thus data flow is not sufficient to order assignments in $\mathbf{proc_2}$ programs as it is for **proc**.

We eschew a deeply formal treatment of this difficulty, which involves complete *operational semantics* for $\mathbf{proc_2}$, e.g. rewrite rules for terms or a machine model. From these we could ascertain which assignment determines the value of each array element as evaluation proceeds. Instead, we specify assignment order directly in terms of $\mathbf{proc_2}$ syntax as a *fait accompli.* It is a tedious but straightforward exercise to show this order is representative of many underlying operational models.

A few terms, definitions, and conventions are required. We write $a[\alpha]$ for a scalar element of a; α stands for an appropriate sequence of natural numbers, and $I(a)$ is the set of all such sequences. Thus $\{a[\alpha] \mid \alpha \in I(a)\}$ is the set of all scalar values in a. Further, ξ, ρ, ψ continue to denote fragments of **proc** (and now $\mathbf{proc_2}$) so that $[\xi]a[\rho]$ denotes some subarray of a. For a generic subarray we write $[\cdot]a[\cdot]$. Recall the leading brackets are for a permutation part, which is omitted when unnecessary. A *target reference* to array variable a is a subarray $[\cdot]a[\cdot]$, on the left hand side of $=$ or serving as the return value array of a procedure call or as a formal parameter in the header of a procedure definition. We assume target references have distinct labels and write a superscript over the array variable, e.g. a^k, to distinguish one from another. References maintain their labels through sharing transformations. E.g. $a^k[i]$ transformed by $\cdot\,[[\xi]t[\psi]/a]$ becomes $[\xi]t^k[\psi][i]$. Formal parameters all have the label $\circ$ by convention. Target references assign array element values. For convenience, formal parameters in the procedure header stand for the assignments to arguments performed by the procedure caller. Each element assignment is considered to be an entity called a *definition* or *def*, written $\overleftarrow{a^k[\alpha]}$ to denote the def of $a[\alpha]$ at target reference with label k. Symmetrically, all subarrays that are not target references are *source references*, including the result array in the procedure header, which has the label $\bullet$ by convention. Source references use element values defined by target references. The result array in the procedure header stands for all uses of the result by the caller. Each *use* is considered to be an entity, written $\overrightarrow{a^\ell[\alpha]}$ to denote the use of the value $a[\alpha]$ at the source reference labeled ℓ.

We can now characterize the sets of defs and uses that can possibly arise from a **proc** subterm.

Example 6.3 Consider the **proc** (or $\mathbf{proc_2}$) subterm

$$\underset{i\in 3:n}{\forall}\; t^k[i] = a^\ell[i] + a^m[i-3]$$

The defs arising here are

$$\overleftarrow{t^k[3]}, \ldots, \overleftarrow{t^k[n+2]}$$

The uses are

$$\overrightarrow{a^\ell[3]}, \ldots, \overrightarrow{a^\ell[n+2]},\ \overrightarrow{a^m[0]}, \ldots, \overrightarrow{a^m[n-1]}$$

□

Figures 6.1 and 6.2 more formally describe defs and uses of general $\mathbf{proc_2}$ subterms. The functions *defs* and *uses* each transform a subterm to an expression whose conventional interpretation is the desired set. We use $(\mathit{defs}\,\xi)$ interchangeably for the expression and (when ξ has no free variables) the set it denotes. Moreover, the

$$
\begin{aligned}
\mathit{defs}(f(a_1,\ldots,a_n;r)\ \rhd \xi) &= \mathit{defs}(\xi) \cup \left\{ \overleftarrow{a_i^\circ[\alpha]} \mid \alpha \in I(a_i), 1 \le i \le n \right\} \\
\mathit{defs}\left(\forall_{i \in b:e} \xi \right) &= \bigcup_{i \in b:e} \mathit{defs}(\xi) \\
\mathit{defs}(\text{let } \xi_1 \text{ in } \xi_2) &= \mathit{defs}(\xi_1) \cup \mathit{defs}(\xi_2) \\
\mathit{defs}(\text{if } \xi_1 \text{ then } \xi_2 \text{ else } \xi_3) &= \mathit{defs}(\xi_1) \cup \mathit{defs}(\xi_2) \cup \mathit{defs}(\xi_3) \\
\mathit{defs}(+\, \xi_1\, \xi_2) &= \mathit{defs}(\xi_1) \cup \mathit{defs}(\xi_2) \\
\mathit{defs}(\wedge\, \xi_1\, \xi_2) &= \mathit{defs}(\xi_1) \cup \mathit{defs}(\xi_2) \\
\mathit{defs}\left(\sum_{i \in b:e} \xi \right) &= \bigcup_{i \in b:e} \mathit{defs}(\xi) \\
\mathit{defs}([\cdot]a[\cdot]) &= \emptyset \qquad \text{(scalar source reference)} \\
\mathit{defs}([\xi]a^k[\rho] = \psi) &= \left\{ \overleftarrow{[\xi]a^k[\rho]} \right\} \cup \mathit{defs}(\psi) \\
\mathit{defs}\left(g(t_1,\ldots,t_n;s^k) \right) &= \left\{ \overline{s^k[\alpha]} \mid \alpha \in I(s) \right\}
\end{aligned}
$$

Figure 6.1: Defs for $\mathbf{proc_2}$ subterms.

$$
\begin{aligned}
\mathit{uses}(f(a_1,\ldots,a_n;r)\ \rhd\ \xi) &= \mathit{uses}(\xi) \cup \left\{\overrightarrow{r^{\bullet}[\alpha]} \mid \alpha \in I(r)\right\} \\
\mathit{uses}\left(\bigvee_{i\in b:e} \xi\right) &= \bigcup_{i\in b:e} \mathit{uses}(\xi) \\
\mathit{uses}(\text{let } \xi_1 \text{ in } \xi_2) &= \mathit{uses}(\xi_1) \cup \mathit{uses}(\xi_2) \\
\mathit{uses}(\text{if } \xi_1 \text{ then } \xi_2 \text{ else } \xi_3) &= \mathit{uses}(\xi_1) \cup \mathit{uses}(\xi_2) \cup \mathit{uses}(\xi_3) \\
\mathit{uses}(+\, \xi_1\, \xi_2) &= \mathit{uses}(\xi_1) \cup \mathit{uses}(\xi_2) \\
\mathit{uses}(\wedge\, \xi_1\, \xi_2) &= \mathit{uses}(\xi_1) \cup \mathit{uses}(\xi_2) \\
\mathit{uses}\left(\sum_{i\in b:e} \xi\right) &= \bigcup_{i\in b:e} \mathit{uses}(\xi) \\
\mathit{uses}([\xi]a^{\ell}[\rho]) &= \left\{\overrightarrow{[\xi]a^{\ell}[\rho]}\right\} \text{ (scalar source reference)} \\
\mathit{uses}([\xi]a[\rho] = \psi) &= \mathit{uses}(\psi) \\
\mathit{uses}\left(g(t_1^{\ell 1},\ldots,t_n^{\ell n};s)\right) &= \left\{\overrightarrow{t_i^{\ell i}[\alpha]} \mid \alpha \in I(t_i^{\ell i}), 1 \le i \le n\right\}
\end{aligned}
$$

Figure 6.2: Uses for **proc** subterms.

sets are presumed closed for subarray equivalence, e.g. if $\overleftarrow{a^k[2]}$ is in $(\mathit{defs}\,\xi)$, then so are $\overleftarrow{a^k[1{:}d_0(a)-1][1]}$ and $\overleftarrow{a^k[2{:}d_0(a)-2][0]}$. If $\overleftarrow{b^k[2,1]}$ is in $(\mathit{defs}\,\xi)$, then so is $\overleftarrow{[1,0]b^k[1,2]}$. The equations take the union of defs and uses for the branches of conditionals, so defs and uses represent a superset of the array elements actually defined and used when the program runs. Similarly, *uses* includes all elements of procedure call arguments when, in fact, the called procedure may ignore some or all.

Defs and uses also express the actions of the procedure caller. The first equation in Figure 6.1 gives the implicit defs of formal parameters by the caller. The first equation in Figure 6.2 gives the implicit uses of the procedure result. For succinctness, we often write e.g. $(\mathit{uses}\,f)$ instead of $\mathit{uses}(f(a_1,\ldots,a_n;r)\ \triangleright\ \xi)$. When we discuss interprocedural sharing in Section 6.8, we will write e.g. $\mathit{uses}\,f(t_1,\ldots,t_n;s)$ to denote the set of uses (and respectively for defs) that arise from f when the parameters $a_1,\ldots,a_n,r$ are bound to arguments $t_1,\ldots,t_n,s$. It is important to realize, however, that def and use sets do not depend on the *values* of these arguments, but only on their dimensions.

To specify which of multiple defs is "seen" by a given use, we next impose a partial order $\ll_f$ (omitting the subscript when it isn't important) on the defs and uses of f. Figure 6.3 gives a function $\mathcal{B}$ that transforms a function definition f without procedure calls to a set expression for the $\ll_f$ relation. It uses the abbreviation $(\mathit{ud}\,\xi)$ for $(\mathit{defs}\,\xi)\cup(\mathit{uses}\,\xi)$. Further, if X and Y are sets containing defs and uses, then $X \ll Y$ stands for $\{x \ll y \mid x \in X, y \in Y\}$.

Intuitively, $\ll$ characterizes precedence of uses and defs during evaluation. Equation (6.5) closes the relation for the body and puts implicit defs of formal parameters before uses and defs in the procedure body, which in turn are before implicit uses of the procedure result. Equations (6.6) and (6.10) gather precedence pairs due to loop iterations. They impose no order between different iterations. Equation (6.7)

$$\begin{aligned}
\mathcal{B}(f(a_1,\ldots,a_n;r)\ \triangleright \xi) &= \Big(\mathcal{B}(\xi) \cup (\{\overleftarrow{a_i^\circ[\alpha]} \mid \alpha \in I(a_i), 1 \le i \le n\} \ll ud(\xi)) \\
&\qquad \cup (ud(\xi) \ll \{\overrightarrow{r^\bullet[\alpha]} \mid \alpha \in I(r)\})\Big)^* && (6.5) \\
\mathcal{B}\big(\underset{i\in b:e}{\forall} \xi\big) &= \bigcup_{i\in b:e} \mathcal{B}(\xi) && (6.6) \\
\mathcal{B}(\text{let } \xi_1 \text{ in } \xi_2) &= (ud(\xi_1) \ll ud(\xi_2)) \cup \mathcal{B}(\xi_1) \cup \mathcal{B}(\xi_2) && (6.7) \\
\mathcal{B}(\text{if } \xi_1 \text{ then } \xi_2 \text{ else } \xi_3) &= (ud(\xi_1) \ll ud(\xi_2, \xi_3)) \\
&\qquad \cup \mathcal{B}(\xi_1) \cup \mathcal{B}(\xi_2) \cup \mathcal{B}(\xi_3) && (6.8) \\
\mathcal{B}(* \, \xi_1 \, \xi_2) &= \mathcal{B}(\xi_1) \cup \mathcal{B}(\xi_2) \qquad (* \text{ is } + \text{ or } \wedge) && (6.9) \\
\mathcal{B}\big(\sum_{i\in 0:e} \xi\big) &= \bigcup_{i\in 0:e} \mathcal{B}(\xi) && (6.10) \\
\mathcal{B}([\cdot]a^\ell[\cdot]) &= \emptyset \qquad \text{(scalar source reference)} && (6.11) \\
\mathcal{B}([\xi]a^k[\rho] = \psi) &= ud(\psi) \ll \left\{\overleftarrow{[\xi]a^k[\rho]}\right\} && (6.12)
\end{aligned}$$

Figure 6.3: Intra-procedural use and def ordering for $\mathbf{proc_2}$.

puts all uses and defs in the head of a let before those in the body. Equation (6.8) puts all uses and defs in the condition before those in the branches. Equation (6.9) accumulates pairs from its arguments. It imposes no order on argument evaluation. Equation (6.11) shows that source references impose no order. Equation (6.12) puts all uses on the right hand side of assignments before the def on the left.

Similarly, if $v_1, v_2 \in (ud\ f)$, we say $v_1 \asymp_f v_2$ if v_1 and v_2 arise in opposite branches of the same conditional.

We now state a condition sufficient to give useful semantics to a $\mathbf{proc}_2$ procedure f.

Definition 6.4 *The def $\overleftarrow{a^k[\alpha]}$ defines the value of the use $\overrightarrow{a^\ell[\alpha]}$ in f if (and only if) both the following hold:*

(a) Definition occurs before use, i.e. $\overleftarrow{a^k[\alpha]} \ll \overrightarrow{a^\ell[\alpha]}$.

(b) Other defs of $a[\alpha]$ are not interposed between the def and use of interest, i.e. for all $\overleftarrow{a^m[\alpha]} \in (defs\ f)$, $m \neq k$, we have

$$\overleftarrow{a^m[\alpha]} \ll \overleftarrow{a^k[\alpha]} \quad \text{or} \quad \overleftarrow{a^m[\alpha]} \asymp \overleftarrow{a^k[\alpha]} \quad \text{or} \quad \overrightarrow{a^\ell[\alpha]} \ll \overleftarrow{a^m[\alpha]} \quad \text{or} \quad \overrightarrow{a^\ell[\alpha]} \asymp \overleftarrow{a^m[\alpha]}$$

When this is true, we write

$$\overleftarrow{a^k[\alpha]} \overset{f}{\rightsquigarrow} \overrightarrow{a^\ell[\alpha]}$$

□

Thus for Example 6.2, (6.7) implies that each assignment of form $x[i] = x[i] + 1$ defines the value of the use in $r[i] = x[i]$. From (6.5), the implicit def of formal parameters defines the use of $x[i] = x[i] + 1$ but not $r[i] = x[i]$, so the problematic evaluation order

$$r[0] = x[0],\ x[0] = x[0] + 1,\ r[1] = x[1],\ x[1] = x[1] + 1, \ldots$$

no longer has right hand sides defined.

6.4 Extensional Equivalence

The next step is to distinguish correct sharing transformations from incorrect ones.

Definition 6.5 *Let f be a* **proc$_2$** *procedure $f(a_1, \ldots, a_n; r) \;\triangleright\; \xi$, and let $f' = f\,[b/a]$. We say f' is* extensionally equivalent *to f (i.e. $f' \equiv f$) if for all arguments $t_1, \ldots, t_n$ of suitable dimensions, the procedure call $f(t_1, \ldots, t_n; s)$ terminates if and only if $f'(t_1, \ldots, t_n; s')$ does, and on termination $s = s'$. Furthermore, if $f' \equiv f$, then $f\,[b/a]$ is* correct. □

Note f' may change the value of an argument array that f does not, as in (6.4).

Unsurprisingly, a sharing transformation is correct if it preserves the definitions seen by all uses.

Theorem 6.6 (Sharing correctness) *Let $f' = f\,[b/a]$. Then $f' \equiv f$ if the defs of a and b elements that define values of uses in f define the values of the same uses in f', i.e.*

$$\begin{aligned} \forall \alpha, k, \ell \;.\; \overleftarrow{a^k[\alpha]} \overset{f}{\rightsquigarrow} \overrightarrow{a^\ell[\alpha]} \quad \text{implies} \quad \overleftarrow{b^k[\alpha]} \overset{f'}{\rightsquigarrow} \overrightarrow{b^\ell[\alpha]} \\ \text{and} \quad \forall \alpha, k, \ell \;.\; \overleftarrow{b^k[\alpha]} \overset{f}{\rightsquigarrow} \overrightarrow{b^\ell[\alpha]} \quad \text{implies} \quad \overleftarrow{b^k[\alpha]} \overset{f'}{\rightsquigarrow} \overrightarrow{b^\ell[\alpha]} \end{aligned} \tag{6.13}$$

Proof. The proof is by elementary structural induction over the code of f and f'. □

6.5 The Sharing Condition

We can now write a sufficient condition for correct sharing transformations.

Theorem 6.7 *We have $f \equiv f\,[b/a]$ if all the defs and uses of each element of a are complete before any defs of the corresponding element of b; i.e.*

$$\forall \overleftrightarrow{a^k[\alpha]} \in (\mathit{udf}), \overleftarrow{b^\ell[\alpha]} \in (\mathit{defs}\ f) \;.\; (\overleftrightarrow{a^k[\alpha]} \ll_f \overleftarrow{b^\ell[\alpha]} \text{ or } \overrightarrow{a^k[\alpha]} \lessgtr_f \overleftarrow{b^\ell[\alpha]}) \tag{6.14}$$

where $\overleftrightarrow{a^k[\alpha]}$ is any use or def from (udf). Alternatively, $f \equiv f\,[b/a]$ if all defs and uses of each element of b are complete before any defs of the corresponding element *of a:*

$$\forall \overleftrightarrow{b^\ell[\alpha]} \in (udf), \overleftarrow{a^k[\alpha]} \in (\textit{defs}\ f) \ . \ (\overleftrightarrow{b^\ell[\alpha]} \ll_f \overleftarrow{a^k[\alpha]} \text{ or } \overleftrightarrow{b^\ell[\alpha]} \gtrless_f \overleftarrow{a^k[\alpha]}) \qquad (6.15)$$

Proof. We will show only that (6.14) implies $f \equiv f\,[b/a]$; (6.15) is symmetric. It is sufficient to show (6.14) implies (6.13), where $f' = f\,[b/a]$.

Consider the first clause of the conjunction in (6.13). Part (a) of Definition 6.4 requires that

$$\overleftarrow{b^k[\alpha]} \ll_{f'} \overrightarrow{b^\ell[\alpha]} \quad \text{when} \quad \overleftarrow{a^k[\alpha]} \ll_f \overrightarrow{a^\ell[\alpha]}$$

This follows from an easily verified property of the definition of $\ll$:

$$\mathcal{B}(f\,[b/a]) = \mathcal{B}(f)\,[b/a] \qquad (6.16)$$

Part (b) of Definition 6.4 requires that if for all $\overleftarrow{a^m[\alpha]} \in (\textit{defs}\ f)$, $m \neq k$,

$$\overleftarrow{a^m[\alpha]} \ll_f \overleftarrow{a^k[\alpha]} \quad \text{or} \quad \overleftarrow{a^m[\alpha]} \gtrless_f \overleftarrow{a^k[\alpha]} \quad \text{or} \quad \overrightarrow{a^\ell[\alpha]} \ll_f \overleftarrow{a^m[\alpha]} \quad \text{or} \quad \overrightarrow{a^\ell[\alpha]} \gtrless_f \overleftarrow{a^m[\alpha]}$$

then it is also true that for all $\overleftarrow{b^m[\alpha]} \in (\textit{defs}\ f')$, $m \neq k$,

$$\overleftarrow{b^m[\alpha]} \ll_{f'} \overleftarrow{b^k[\alpha]} \quad \text{or} \quad \overleftarrow{b^m[\alpha]} \gtrless_{f'} \overleftarrow{b^k[\alpha]} \quad \text{or} \quad \overrightarrow{b^\ell[\alpha]} \ll_{f'} \overleftarrow{b^m[\alpha]} \quad \text{or} \quad \overrightarrow{b^\ell[\alpha]} \gtrless_{f'} \overleftarrow{b^m[\alpha]} \qquad (6.17)$$

Consider that $\overleftarrow{b^m[\alpha]} \in (\textit{defs}\ f')$ fall naturally into two classes:

1. Those where $\overleftarrow{a^m[\alpha]} \in (\textit{defs}\ f)$, i.e. those defs of b's elements in f' corresponding to defs of a's elements in f. Here, (6.17) follows by (6.16).

2. Those where $\overleftarrow{b^m[\alpha]} \in (\textit{defs}\ f)$, which are defs of b's elements in f' that are also in f. Here (6.14) implies

$$\forall \overleftrightarrow{b^k[\alpha]} \in (udf'), \overleftarrow{b^\ell[\alpha]} \in (\textit{defs}\ f') \ . \ (\overleftrightarrow{b^k[\alpha]} \ll_{f'} \overleftarrow{b^\ell[\alpha]} \text{ or } \overrightarrow{b^k[\alpha]} \gtrless_{f'} \overleftarrow{b^\ell[\alpha]}) \qquad (6.18)$$

again by (6.16), which yields the desired result because it implies

$$\overrightarrow{b^\ell[\alpha]} \ll_{f'} \overleftarrow{b^m[\alpha]}$$

The second clause of the conjunction is similar to the first. Part (a) of Definition 6.4 requires

$$\overleftarrow{b^k[\alpha]} \ll_{f'} \overrightarrow{b^\ell[\alpha]} \quad \text{when} \quad \overleftarrow{b^k[\alpha]} \ll_f \overrightarrow{b^\ell[\alpha]}$$

which is immediate.

Part (b) of Definition 6.4 requires that if for all $\overleftarrow{b^m[\alpha]} \in (\mathit{defs}\ f)$, $m \neq k$,

$$\overleftarrow{b^m[\alpha]} \ll_f \overleftarrow{b^k[\alpha]} \quad \text{or} \quad \overleftarrow{b^m[\alpha]} \not\asymp_f \overleftarrow{b^k[\alpha]} \quad \text{or} \quad \overrightarrow{b^\ell[\alpha]} \ll_f \overleftarrow{b^m[\alpha]} \quad \text{or} \quad \overrightarrow{b^\ell[\alpha]} \not\asymp_f \overleftarrow{b^m[\alpha]}$$

then it is also true that for all $\overleftarrow{b^m[\alpha]} \in (\mathit{defs}\ f')$, $m \neq k$,

$$\overleftarrow{b^m[\alpha]} \ll_{f'} \overleftarrow{b^k[\alpha]} \quad \text{or} \quad \overleftarrow{b^m[\alpha]} \not\asymp_{f'} \overleftarrow{b^k[\alpha]} \quad \text{or} \quad \overrightarrow{b^\ell[\alpha]} \ll_{f'} \overleftarrow{b^m[\alpha]} \quad \text{or} \quad \overrightarrow{b^\ell[\alpha]} \not\asymp_{f'} \overleftarrow{b^m[\alpha]}$$

The same two classes of defs $\overleftarrow{b^m[\alpha]} \in (\mathit{defs}\ f')$ are important here. For 1. above, (6.18) implies $\overrightarrow{b^\ell[\alpha]} \not\asymp_{f'} \overleftarrow{b^m[\alpha]}$; for 2. the result follows from (6.16). □

We need a stronger version of this theorem for the general case of a $\mathbf{proc_2}$ procedure $f^{(n)}$ obtained by n sharing transformations of a **proc** procedure $f^{(0)}$. If x is some variable occurring in $f^{(n)}$ (and therefore $f^{(0)}$), let *ancestors*(x) be the set of variables in $f^{(0)}$ sharing storage with x in $f^{(n)}$ (including x). Further let *labels*(y, f) be the set of reference labels associated with variable y in f.

Corollary 6.8 *Let $f^{(0)}$ be a* **proc** *procedure and for any $n \geq 0$, let $f^{(i+1)} = f^{(i)}\,[b_i/a_i]$ for all $0 \leq i < n$, where each of these sharing transformations is correct by recursive application of this corollary. Finally let $f^{(n+1)} = f^{(n)}\,[b/a]$. Then $f^{(n+1)} \equiv f^{(n)}$ if for all variable pairs (x, y) such that $x \in$ ancestors(a) and $y \in$ ancestors(b) we have*

$$\begin{array}{l} \forall k \in \mathit{labels}(x, f^{(0)}), \ell \in \mathit{labels}(y, f^{(0)})\ . \\ \quad \forall \overleftrightarrow{a^k[\alpha]} \in (\mathit{ud} f), \overleftarrow{b^\ell[\alpha]} \in (\mathit{defs}\ f)\ .\ (\overleftrightarrow{a^k[\alpha]} \ll \overleftarrow{b^\ell[\alpha]} \text{ or } \overleftrightarrow{a^k[\alpha]} \not\asymp \overleftarrow{b^\ell[\alpha]}) \end{array} \tag{6.19}$$

or else

$$\begin{array}{l} \forall k \in \mathit{labels}(x, f^{(0)}), \ell \in \mathit{labels}(y, f^{(0)})\ . \\ \quad \forall \overleftrightarrow{b^\ell[\alpha]} \in (\mathit{ud} f), \overleftarrow{a^k[\alpha]} \in (\mathit{defs}\ f)\ .\ (\overleftrightarrow{b^\ell[\alpha]} \ll \overleftarrow{a^k[\alpha]} \text{ or } \overleftrightarrow{b^\ell[\alpha]} \not\asymp \overleftarrow{a^k[\alpha]}) \end{array} \tag{6.20}$$

Proof. The proof is similar to Theorem 6.7 with the additional observation that if $\overleftarrow{a^k[\alpha]} \overset{f^{(n)}}{\leadsto} \overrightarrow{a^\ell[\alpha]}$, then k and ℓ must label the same variable in $f^{(0)}$. Moreover, if $\overleftarrow{b^{k'}[\alpha]} \overset{f^{(n)}}{\leadsto} \overrightarrow{b^{\ell'}[\alpha]}$ then k' and ℓ' again label the same variable in $f^{(0)}$, but that variable is different from the one labeled k and ℓ. □

This corollary removes constraints on the order of successive sharing transformations imposed by Theorem 6.7. For instance, (6.3) is correct by Corollary 6.8, but not by Theorem 6.7. The problem in the latter case is that x is defined by both the first and last operations (with respect to $\ll$) of the procedure body. Hence Theorem 6.7 is inherently powerless to prove correctness of sharing involving x. On the other hand, Corollary 6.8 exploits that fact the all uses and defs of t are "between" (again with respect to $\ll$) all the uses and defs arising from x in g and all those arising from r in g. Thus defs of t cannot interfere with any values seen by uses of x and *vice versa.*

6.6 A Syntactic Sharing Condition

Corollary 6.8 leads directly to an algorithm. Let S be a predicate on a pair of syntactic references with labels k and ℓ such that

$$S(a^k, b^\ell, f) \text{ implies } \forall \overleftrightarrow{a^k[\alpha]} \in (ud\, f), \overleftarrow{b^\ell[\alpha]} \in (\mathit{defs}\ f)\ .\ (\overleftrightarrow{a^k[\alpha]} \ll \overleftarrow{b^\ell[\alpha]} \text{ or } \overleftrightarrow{a^k[\alpha]} \between \overleftarrow{b^\ell[\alpha]})$$

That is, S employs the syntax of a pair of references to establish, insofar as those two references are concerned, that sharing storage for a and b is correct. To fully establish correctness, the algorithm of Figure 6.4 considers all pairs of such references. It returns *correct* if the sharing transformation $f\,[b/a]$ is correct, and *fail* otherwise.

It remains to develop a useful S. Since sharing is advantageous, it should be as strong (true on as many arguments) as possible.

Theorem 6.9 (Disjoint ranges) *Consider the sets*

$$I^k = \{\alpha \mid \overleftrightarrow{a^k[\alpha]} \in ud(f)\} \quad \text{and} \quad I^\ell = \{\alpha \mid \overrightarrow{b^\ell[\alpha]} \in uses(f)\}$$

Algorithm: Correctness of sharing transformation $f\,[b/a]$

Inputs: $\mathbf{proc_2}$ procedure f, array variable a, and subarray b in f.

Output: One of $\{correct, fail\}$

Method: If for all variable pairs x, y such that
$x \in ancestors(a)$ and $y \in ancestors(b)$ we have
$\forall\, k \in labels(x), \ell \in labels(y)\,.\, S(a^k, b^\ell, f)$ or
$\forall\, k \in labels(x), \ell \in labels(y)\,.\, S(b^\ell, a^k, f)$,
then return *correct.*
Else return *fail.*

Figure 6.4: Algorithm for sharing correctness.

If $I^k \cap I^\ell = \emptyset$, then $S(a^k, b^\ell, f)$.

Proof. In this case the quantifier in the definition of S is vacuously true. □

We can determine when this theorem holds by examining the syntax of references k and ℓ and the iteration ranges of loops whose variables occur in the references. This often involves submatrix algebra. For instance, suppose

$$b^\ell = [1, 0]t^\ell[1 : n, 0 : n]$$

and f contains references

$$t^\ell[i_0, 1] \quad \text{and} \quad a^k[i_1, i_0]$$

where ℓ is a target reference. We can rewrite ℓ as

$$([1, 0]t^\ell[1{:}n, 0{:}n])[0, i_0]$$

so we have $S(a^k, b^\ell, f)$ as long as the iteration range of i_1 does not include zero. The simplest cases of the theorem are summarized in the following.

Corollary 6.10 (Irrelevant references) *Suppose k and ℓ are reference labels in f, and a and b are subarrays of array variables in f. Then if reference k is to an array* *variable other than that of a or if reference ℓ is to an array variable other than that of b, or if ℓ labels a source reference, then* $S(a^k, b^\ell, f)$.

The other cases arise from the definition of Figure 6.3. In particular, we seek theorems of the following flavor.

Theorem 6.11 (Let) *Suppose the body of f is of form*

$$\xi \,\mathrm{let}\, \psi \,\mathrm{in}\, \rho$$

If k is a reference label in ψ and ℓ a target reference label in ρ and ξ contains no loops with variables occurring in these references, then $S(a^k, b^\ell, f)$.

Proof. From (6.7), if the references involve no variables of surrounding loops, then all uses and defs of the *let* header are related by $\ll$ to all defs of the body. □

Likewise, from (6.12) we obtain the following due to the order imposed by $=$.

Theorem 6.12 (Assignment) *Suppose the body of f is of form*

$$\xi \, b^\ell[\xi_b] = \cdots a^k[\xi_a] \cdots$$

where $\xi_b = \xi_a$. Then $S(a^k, b^\ell, f)$.

Proof. Follows from (6.12). □

Applying this theorem may again involve elementary submatrix algebra. For instance, suppose $b^\ell = [1,0]t^\ell[1:n, 0:n]$ and f is of the form

$$\cdots t^\ell[i_0, i_1 + 1] = a^k[i_1, i_0]$$

This is the same as

$$\cdots ([1,0]t^\ell[1{:}n, 0{:}n])[i_1, i_0] = a^k[i_1, i_0]$$

so we have $S(a^k, b^\ell, f)$.

The definition of $\lessgtr$ leads us similarly to the following.

Inputs: $\mathbf{proc_2}$ procedure f, subarrays a and b, a reference labeled k, target reference labeled ℓ

Output: A value suitable for use as $S(a^k, b^\ell, f)$.

Method: Test the body syntax of f to determine if the conditions of Theorem 6.9, 6.11, 6.12, 6.13, 6.14, or 6.15 are met. If so, return *true*. Otherwise, return *false*.

Figure 6.5: Algorithm for $S(a^k, b^\ell, f)$.

Theorem 6.13 (Conditional) *Suppose the body of f is of form*

$$\xi \,\mathrm{if}\, \psi \,\mathrm{then}\, \rho_T \,\mathrm{else}\, \rho_F$$

If k and ℓ label a reference and target reference respectively, where (a) k occurs in ρ_T and ℓ in ρ_F or vice versa, *or else (b) k occurs in ψ and ℓ occurs in either ρ_T or ρ_F, and neither reference involves variables of loops in ξ, then $S(a^k, b^\ell, f)$.*

Proof. If references k and ℓ do not depend on surrounding loops, then all pairs of uses and defs arising from them, one from each, are related by $\lessgtr$ in case (a) and $\ll$ in case (b) by (6.8). □

Another fundamental case is this.

Theorem 6.14 (Parameter–result) *For all reference labels k and subarrays a and b, we have $S(a^\circ, b^k)$ and $S(a^k, b^\bullet)$.*

Proof. Follows from (6.5). □

The algorithm of Figure 6.5 employs Theorems 6.9, 6.11, 6.12, 6.13, and 6.14 (also 6.15 given later) to compute a suitable S.

6.7 Strengthening $\ll$

The definition of $\ll$ given in Figure 6.3 is consistent with parallel evaluation of e.g. the left and right hand side of $\wedge$. In general, we obtain more opportunities for sharing if the **proc$_2$** machine operates sequentially. For instance, the **svexp** fragment

$$\begin{array}{c}\mathrm{Join}\,0\,(\mathrm{Extr}_0^0\,0\,\mathrm{Extr}_1^0\,0\,x)\\ (\mathrm{Extr}_1^0\,0\,x)\end{array} \tag{6.21}$$

might compile to (with $x : n$ and $r : n$)

$$r[0] = x[1] \wedge \forall_{i \in 1:n-1}\, r[i] = x[i]$$

Sharing of r and x is not correct by the definition of $\ll$ in Figure 6.3 and resulting definition of S. The right hand side may define element 1 of the shared vector before the left side uses it. However, if we strengthen $\ll$ so that uses and defs of the left hand side of $\wedge$ are before (with respect to $\ll$) the right hand side, then sharing is correct. Similar opportunities for sharing are exposed by ordering the iterations of $+$, $\forall$, and $\sum$.

There is in general no *a priori* reason to prefer one order over another in these cases. For instance, a symmetric variant of (6.21) admits sharing only if $\wedge$ evaluates its *right* operand first. In outline, then, a useful extension of the algorithm presented here determines evaluation order as a side-effect of applying S. That is, we modify S to return both its predicate value and a *caveat* that imposes an order of operand evaluation on an occurrence of $+$, $\wedge$, $\forall$, or $\sum$. Caveats are incorporated in the final compiled code, which is necessarily an extension of **proc$_2$** with sequential versions of these operations.

Of course, mutually contradictory caveats can arise. Consider an alternative vector reverse code that swaps the end-most elements and calls itself recursively to swap the rest.

$$\begin{aligned} rev\ x \equiv\ & \text{if } (\text{NumOf } 0\ x) \leq 1 \text{ then } x \\ & \text{else Join } 0\, (\text{Extr}_0^1\, 0\, x) \\ & \qquad\qquad \text{Join } 0\, (rev\ \text{Extr}_1^0\, 0\ \text{Extr}_1^1\, 0\, x) \\ & \qquad\qquad\qquad (\text{Extr}_0^0\, 0\, x) \end{aligned}$$

This compiles to

$$\begin{aligned} rev(x; r)\ \rhd\ & \text{if } n \leq 1 \text{ then } \underset{i \in 0:n}{\forall}\ r[i] = x[i] \\ & \text{else } r[0] = x[n-1] \wedge rev(x[1{:}n-2]; r[1{:}n-2]) \wedge r[n-1] = x[0] \end{aligned}$$

where $x : n$. There is no order of $\wedge$ operand evaluation that will satisfy the caveats resulting from $r[0] = x[n-1]$ and $r[n-1] = x[0]$ to allow sharing of x and r. However, observe that sharing of these vectors is possible at the cost of only a scalar copy. That is, the compiler can eliminate the use of e.g. $x[0]$ in the last $\wedge$ operand by copying $x[0]$ to a "temporary:"

$$\begin{aligned} rev(x; r)\ \rhd\ & \text{if } n \leq 1 \text{ then } \underset{i \in 0:n}{\forall}\ r[i] = x[i] \\ & \text{else let } t\ \rhd\ t = x[0] \\ & \qquad \text{in } r[0] = x[n-1] \wedge rev(x[1{:}n-2]; r[1{:}n-2]) \wedge r[n-1] = t \end{aligned}$$

Now a caveat that $r[0] = x[n-1]$ be evaluated before $r[n-1] = t$ is sufficient for correctness of the following sharing transformation:

$$\begin{aligned} rev(x; x)\ \rhd\ & \text{if } n \leq 1 \text{ then } \{\textit{do nothing}\} \\ & \text{else let } t\ \rhd\ t = x[0] \\ & \qquad \text{in } x[0] = x[n-1] \wedge rev(x[1{:}n-2]; x[1{:}n-2]) \wedge x[n-1] = t \end{aligned}$$

If the compiler notes that the recursive call to *rev* can be executed last to make the procedure tail-recursive, then this code is as good as any we can write.

6.8 Interprocedural Sharing Analysis

Recall that in Figure 6.3 we omitted the definition of $\ll$ for uses and defs arising respectively from the arguments and result of a procedure call. Thus the discussion above is restricted *de facto* to *intra-procedural* sharing: if a and b appear anywhere in f as argument and result of the same procedure call, we are not yet equipped to decide if $f\,[b/a]$ is correct.

Fortunately, there is nothing in the sharing condition and associated algorithms that impairs them for inter-procedural sharing. We need only specify $\ll$ for procedure calls. Trivially, we could supplement Figure 6.3 with

$$\mathcal{B}(g(t_1^{k_1}, \ldots, t_n^{k_n}; s^\ell)) = \emptyset \tag{6.22}$$

but this says only that the sharing transformations involving any of the t_i and s are incorrect. As in Section 6.7, we want to make $\ll$ as strong as possible to allow sharing as often as possible.

6.8.1 The nonrecursive case

What *can* be said of the order of uses and defs arising from a procedure call? Begin with the the non-recursive case. Since $\mathbf{proc_2}$ arguments are passed by reference, the arguments t_i and result s of a call as in (6.22) are identified with parameters a_i and return array r respectively in the body of g. Thus we can usefully say (and given operational semantics for $\mathbf{proc_2}$ we would prove) that

$$\overrightarrow{t_i^{k_i}[\alpha_t]} \ll_f \overleftarrow{s^\ell[\alpha_s]}$$

whenever all uses of $a_i[\alpha_t]$ are before all defs of $r[\alpha_s]$ in g, i.e.

$$\begin{aligned}
&\forall \overrightarrow{a_i^{k_i}[\alpha_t]} \in \mathit{uses}\, g(t_1, \ldots, t_n; s), \overleftarrow{r^\ell[\alpha_s]} \in \mathit{defs}\, g(t_1, \ldots, t_n; s)\,. \\
&\qquad\qquad\qquad\qquad \overrightarrow{a_i^{k_i}[\alpha_t]} \ll_g \overleftarrow{r^\ell[\alpha_s]} \quad \text{or} \quad \overrightarrow{a_i^{k_i}[\alpha_t]} \not\asymp_g \overleftarrow{r^\ell[\alpha_s]}
\end{aligned}$$

The appropriate supplement to Figure 6.3 is thus

$$\begin{aligned}
&\mathcal{B}(g(t_1^{k_1}, \ldots, t_n^{k_n}; s)) = \\
&\qquad \{\overrightarrow{t_i^{k_i}[\alpha_t]} \ll_f \overleftarrow{s^\ell[\alpha_s]} \mid
\end{aligned}$$

$$\forall \overrightarrow{a_i^{k_i}[\alpha_t]} \in uses\, g(t_1, \ldots, t_n; s), \overleftarrow{r^\ell[\alpha_s]} \in defs\, g(t_1, \ldots, t_n; s)\,.$$
$$\overrightarrow{a_i^{k_i}[\alpha_t]} \ll_g \overleftarrow{r^\ell[\alpha_s]} \text{ or } \overrightarrow{a_i^{k_i}[\alpha_t]} \not\gg_g \overleftarrow{r^\ell[\alpha_s]}\} \tag{6.23}$$

In essence this says that an argument array of a procedure call in f is sharable with the corresponding return value array in the same manner that the corresponding parameter in g is sharable with g's result array. This provides a useful extension for S.

Theorem 6.15 (Argument–return) *Suppose the body of f is of the form*

$$\xi\, g(\ldots, a^k[\xi_a], \ldots; b^\ell[\xi_b])$$

with g defined as $g(\ldots, x, \ldots; r) \;\triangleright\, \cdot$, where parameter x corresponds to argument a. Further suppose there exists a subarray functional $\cdot[\psi]$ such that

$$a[\xi_a][\psi] = a[\xi_b] \tag{6.24}$$

Then $S(a^k, b^\ell, f)$ if $\forall\, k, \ell\,.\, S(x^k[\psi], r^\ell, g)$. Similarly, if there exists a subarray functional $\cdot[\rho]$ such that $b[\xi_b][\rho] = b[\xi_a]$, then $S(a^k, b^\ell, f)$ if $\forall\, k, \ell\,.\, S(x^k, r^\ell[\rho], g)$.

Proof. If $\forall\, k, \ell\,.\, S(x^k[\psi], r^\ell, g)$, then by (6.23), $S(a^k[\xi_a][\psi], b^\ell[\xi_b], f)$ and from (6.24), $S(a^k[\xi_b], b^\ell[\xi_b], f)$. Since this relates all use–def pairs $\overrightarrow{a^k[\alpha]}, \overleftarrow{b^\ell[\alpha]}$ in $ud(f)$, we also have $S(a^k,\, b^\ell, f)$ as desired by the definition of S. The other case is symmetric. □

6.9 The Recursive Case

It is fortuitous that equation (6.23) and Theorem 6.15, which we obtained by considering non-recursive calls, remain useful in the recursive case, where $g = f$. Here we have procedure f defined by $f(a_1, \ldots a_n; r) \;\triangleright\, \cdot$ containing calls of the form $f(t_1, \ldots, t_n; s)$. Examining Equation (6.23) and Theorem 6.15, we find they remain meaningful except when for some i, t_i is a subarray of a_i and also s is a subarray of r, which makes them circular.

The limited nature of this exception is important because the **proc** compiler, *sans* the simple copy elimination optimization of Section 5.5.1, never returns code where this occurs. That is, every procedure call is of the form

$$\begin{array}{l} \text{let } t_1 \ \rhd \xi_1 \wedge \cdots \wedge t_n \ \rhd \xi_n \\ \text{in let } s \ \rhd f(t_1, \ldots, t_n; s) \\ \text{in } \cdots \end{array}$$

where the t_i and s do not appear elsewhere in the procedure. Thus, the final algorithm orders transformations so that sharing of *let*-defined argument and result arrays like the t_i and s is performed before sharing of these arrays with parameters and return arrays. In particular, if f can be transformed to operate in place as in (6.1), we can immediately transform all recursive calls to f so that the respective argument and result arrays share storage as in (6.2).

6.10 A Sharing Optimization Algorithm

Where do we stand? The algorithm of Figure 6.4 conservatively indicates when the sharing transformation $f\,[b/a]$ is correct. Furthermore, we have an algorithm to compute a usefully strong S, the required subroutine. A complete sharing optimization algorithm need only compose correct sharing transformations. Let a *sharing prospect for* f be a pair $[b/a]$ where a is an array variable and b is a subarray of the same dimensions involving an array variable in f. Suppose we have an algorithm P that constructs a set of sharing prospects P_f from the syntax of f. Then the function *optimize* in Figure 6.6 is an algorithm that returns the set of all correct compositions of sharing transformations induced by P.

6.11 Deducing prospects

It remains to give an algorithm P for deducing a set P_f of useful sharing prospects from the syntax of a $\mathbf{proc_2}$ procedure f. Here "useful" means "likely to be correct."

Inputs: $\mathbf{proc_2}$ procedure f and sharing prospect algorithm P.

Output: Set of all sharing optimizations of f possible with P.

Method: Return *optimize*(f, P) defined as follows:

$$optimize(f, P) = (C = \emptyset) \rightarrow \{f\}, \quad \bigcup_{[b/a] \in C} optimize(f\,[b/a])$$
$$\text{where } C = \{\,[b/a] \in P_f \mid \text{Fig. 6.4 returns } correct \text{ on } f, a, b\}$$

Figure 6.6: Algorithm returning all optimizations of f given P.

We restrict the search to *in-place* prospects, those that could result in sharing of a source array and the target array of some = or some procedure call in f. More precisely, we seek prospects of the form $[[\cdot]t[\cdot]/s]$ or $[[\cdot]s[\cdot]/t]$ such that s and t appear somewhere at least once on the right and left side, respectively, of the same =, or in the argument and result, respectively, of the same procedure call. While it is not clear that this is the only way to proceed, there do not appear to be useful classes of prospects that are missed by this approach.

6.11.1 Memory allocation

In deciding whether a prospect is useful, we must first decide if it is feasible. The footnote on page 156 discusses one case where a prospect that is otherwise desirable may be ruled out by concerns other than the ordering of uses and defs. Another example is the where the transformation $f\,[b/a]$ is correct by use and def order, but there is no *a priori* relationship between the sizes of the arrays involved in a and b. For example, consider

$$sum(x, y) \equiv \text{Join } 0\ [0]$$
$$\qquad\qquad\qquad \text{Proj } 7\ 0\ ((\ Typ\ 0\ 7\ x) + (\text{Typ } 0\ 7\ y))$$

For $x : m$ and $y : n$, the inner vectors of the result are of length min $\{m, n\} + 1$, the vector sum over the common elements of the inner vectors of the arguments, each with a zero prepended. The compiler returns

$$f(x, y; r) \ \triangleright r[0] = 0 \wedge \underset{i \in 0:\min\{m,n\}}{\forall} r[i+1] = x[i] + y[i] \tag{6.25}$$

Here, as an implementation choice, we can share either x or y with a subarray of r as in the footnote on page 156, or we can arrange for one of the parameters to be allocated with enough space to receive the result (i.e. one extra element), or we can give up on sharing altogether. We abstract such details of the way allocation affects sharing: the predicate *feasible*(a, b, f), is true only if a can possibly share storage with b given the memory allocation scheme in effect.

6.11.2 Prospects due to =

Suppose f is of the form

$$\xi \ t^k[\rho] = s^\ell[\psi] \tag{6.26}$$

Then subarrays $[\xi_1]t[\xi_2]$ such that $S(s^\ell, [\xi_1]t^k[\xi_2])$ by Theorem 6.12 (and symmetrically $[\xi_3]s[\xi_4]$ such that $S([\xi_3]s^\ell[\xi_4], t^k))$ are likely to yield the *only* correct sharing transformations involving variables s and t. Intuitively, this is because the theorem applies only if each assignment to a target element uses only the source element at the same index position to compute the assigned value. If the array and subarray elements do not match in this manner, S is false and the sharing algorithm returns *fail*. The sole exception is when (disregarding the symmetric case until we give the final algorithm) the set

$$\{\alpha \mid \overrightarrow{s^\ell[\alpha]} \in \mathit{uses}(f)\} \cap \{\alpha \mid \overleftarrow{[\xi_1]t^k[\xi_2][\alpha]} \in \mathit{defs}(f)\}$$

is provably empty, whereupon Theorem 6.9 applies. This says that the assignments to $[\xi_1]t[\xi_2]$ miss the uses of s completely due to disjoint iteration ranges.

We are left with the problem of deducing the desired prospects from particular assignments. From the discussion thus far, we see that the Disjoint Iteration Theorem (Theorem 6.9) provides no help. However, Theorem 6.12 allows us to deduce, by simple submatrix algebra, exactly the prospects $[[\xi_1]t[\xi_2]/s]$ such that $S(s^\ell, [\xi_1]t^k[\xi_2])$ by the theorem. Consider the case where s and t in (6.26) are vectors, so ρ and ψ are each a simple indexer. If one contains a loop variable, then the other must have the same for the theorem to apply. In particular, suppose we have

$$\xi t^k[T+i] = s^\ell[S+i]$$

where T and S are the constant parts of the indexers. Then it is easy to see that we want the prospect $[t[T-S{:}d_0(s)]/s]$ if $T \geq S$ and if *feasible*$(s, t[T-S{:}d_0(s)])$ because

$$t[T-S{:}d_0(s)][S+i] = t[0{:}d_0(s)][T+i]$$

Note that *feasible* will have to check that t is long enough, i.e. $d_0(t) \geq T-S+d_0(s)$, or else arrange for t to be allocated with length at least $T-S+d_0(s)$. An example with $T=4$, $S=1$, and $d_0(s)=5$ yielding $[t[3{:}5]/s]$ is shown in Figure 6.7. We also want the symmetric prospect $[s[S-T{:}d_0(t)]/t]$ if $S \geq T$ and if *feasible*$(t, s[S-T{:}d_0(t)])$.

We can generalize this thinking to s and t with arbitrary numbers of dimensions. Consider a source and target reference of the form

$$s[I_0, \ldots, I_n] \quad \text{and} \quad t[J_0, \ldots, J_m]$$

where the I_k and J_ℓ are arbitrary simple indexers. Intuitively, to find in-place prospects of the form $[[\xi_1]t[\xi_2]/s]$, we must find permutations ξ_1 and corresponding indexer lists ξ_2 that make I_k compatible with J_k in the same sense as the vector indexers above for $0 \leq k \leq n$. For this we construct the binary relation $\cdot \propto \cdot$, where $p \propto q$ only if

- $I_p = S_p + i$ and $J_q = T_q + i$ where S_p and T_q are constants such that $S_p \leq T_q$ and i is a loop variable or else

proc$_2$ code:

$\underset{i \in 0:3}{\forall} t[4+i] = s[1+i]$

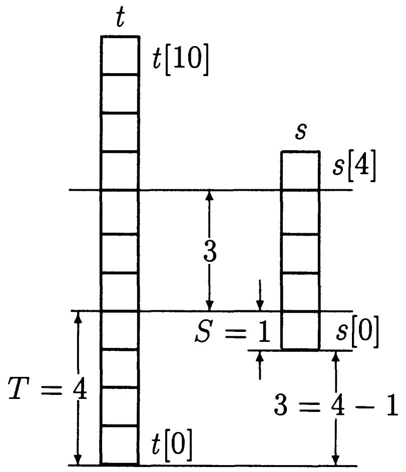


Figure 6.7: In–place prospect $[t[3{:}5]/s]$.

- $I_p = S_p$ and $J_q = T_q$ where S_p and T_q are as above.

This makes precise the notion of "compatibility" above. Let z be a 1–1 map on $0 \leq p \leq n$ such that $p \propto z(p)$. Thus z is a bipartite matching in $\propto$ that covers all the axes of s, and the range of z is a (possibly improper) subset of the axes of t. Let Let $q_0, q_1, \ldots q_n$ be the range of z as a sequence in ascending natural order. Then elements of the desired permutation $\xi_1 = p_0, \ldots, p_m$ are given by

$$p_k = \begin{cases} z(\ell) & \text{if for some } \ell,\ k = q_\ell \\ k & \text{otherwise} \end{cases}$$

and the indexers $\xi_2 = K_0, \ldots, K_m$ are given by

$$K_k = \begin{cases} T_{z(\ell)} - S_\ell{:}d_\ell(s) & \text{if for some } \ell,\ k = q_\ell \\ J_k & \text{otherwise} \end{cases}$$

This are easily verified by submatrix algebra.

The number of bipartite matchings z is liable to be exponential in the number of matrix axes. However, since real programs rarely deal with matrices of more than four or five dimensions, this is not an important problem. Moreover, practical assignments tend to have sparse $\propto$ relationships, hence few matchings.

6.11.3 Prospects due to non-recursive calls

At a non-recursive call $g(\ldots, b^{\ell}, \ldots; a^k)$, where a and b may be subarrays due to earlier transformations, we wish to find opportunities to share storage for the arrays named in a and b. That is, want to establish which, if any, ξ_1, ξ_2 satisfy $S([\xi_1]b^{\ell}[\xi_2], a^k, f)$.[4] From Theorem 6.15, a sufficient condition for this is that g operate in place. That is, for any transformation in *optimize*(g, P) of the form $g^{(i)}(\ldots, x, \ldots; [\xi_1]x[\xi_2]) \;\rhd\; \cdot$, where x is the formal parameter corresponding to b above, we can be sure $S([\xi_1]b^{\ell}[\xi_2], a^k, f)$ at a call of $g^{(i)}$. Hence $[[\xi_1]b[\xi_2]/a]$ is an in–place prospect if a is an array variable. When a is a proper subarray, we solve for the desired prospect by submatrix algebra much as for $=$ in the previous section.

6.11.4 Prospects due to recursive calls

In applying the reasoning of the previous section to recursive calls it should come as no surprise that we encounter a circular relationship: we must know that f transforms to operate in place in order to find prospects so that Figure 6.6 can eventually transform f to operate in place.

Once again, there is a way to unravel the circularity. The key is to note that if f terminates, then along one or more conditional paths through the program, the return array is defined in terms of the parameters by only $=$ and non-recursive procedure calls (and not recursive calls) assigning values. If the non-recursive path (or paths) can be transformed to operate in place, i.e. to correctly evaluate the return value into (a subarray of) one of the parameters, then Theorem 6.15 tells us that at each recursive call, the return array can share storage with the corresponding argument (subarray).[5]

To exploit this observation, we apply *optimize* in Figure 6.6 to f using only

[4]If the sharing described in the footnote of page 156 is permissible, then we are also interested in $S(b^{\ell}, [\xi_3]a^k[\xi_4], f)$.

[5]The same non-recursive path(s) provided Theorem 4.13.

Inputs: $\mathbf{proc}_2$ procedure f.

Outputs: Set P_f of in–place prospects for f.

Method: Let $P_f = E \cup N \cup R$ where E is the set of in–place prospects due to $=$, (Section 6.11.2), N is the set of prospects due to non-recursive calls (Section 6.11.3), and R is the set of prospects due to recursive calls to f (Section 6.11.4).

Figure 6.8: Algorithm P to find in–place prospects.

prospects that have nothing to do with recursive calls. I.e. let P^- be the algorithm that returns for procedure f the set of in–place prospects not involving argument or result arrays of recursive calls. These we already know how to find from occurrences of $=$ and non-recursive procedure calls as in Section 6.11.2 and 6.11.3 respectively. Then for any transformation in *optimize*(f, P^-), of the form

$$f^{(i)}(\ldots, x, \ldots; [\xi_1]x[\xi_2]) \rhd \cdot$$

we can be sure $S([\xi_1]b^\ell[\xi_2], a^k, f)$ for any recursive call $f^{(i)}(\ldots b^\ell \ldots; a^k)$.

6.11.5 Efficiency

To simplify exposition, the algorithm of Figure 6.6 is written as a brute force enumeration; it generates a set of size generally exponential in the size of f. This section considers how *optimize* can be implemented with more reasonable efficiency.

First, there is no need to represent the set *optimize*(f, P) explicitly. It can be enumerated one element at a time.

Next, it is not necessary to execute P "from scratch" for each new transformation. *Post hoc* analysis of algorithm P in Figure 6.8 reveals that $P_{f\,[b/a]}$ is the same as P_f wherein all occurrences of a are replaced by b and all resulting pairs with the variable of b occurring on both sides discarded. Denote this transformation $P_f\,[b/a]$ (i.e. it is a theorem that $P_{f\,[b/a]} = P_f\,[b/a]$).

Inputs: $\mathbf{proc}_2$ procedure f and sharing prospect algorithm P.

Output: Set of all sharing optimizations of f possible with P.

Method: For all permutations U of P_f^-
    let $f' = f$
    while $U \neq \emptyset$
        remove the first prospect $[b/a]$ from U
        if Figure 6.4 returns *correct* on f', a, b
            replace U by $U\,[b/a]$
            replace f' by $f'\,[b/a]$
    if f' operates in place, i.e. $f'(\ldots, a, \ldots, [\xi_1]a[\xi_2]) \rhd \cdot$ then
        for all respective argument–result pairs t, s of recursive calls to f
            replace f' by $f'\,[[\xi_1]t[\xi_2]/s]$.
    for all permutations V of the set of remaining prospects $[b/a]$ in P_f
        while $V \neq \emptyset$
            remove the first prospect $[b/a]$ from V
            if Figure 6.4 returns *correct* on f', a, b
                replace V by $V\,[b/a]$
                replace f' by $f'\,[b/a]$
        output f'

Figure 6.9: Faster algorithm returning all optimizations of f given P.

Finally, as already noted, it is desirable to consider prospects in a sequence that "unravels the recursive knot." From the previous discussion, such an order is—

- Prospects not involving argument or result arrays of recursive procedure calls.
- Prospects involving argument–result pairs for the same recursive procedure call which are certainly correct because the procedure has been transformed to operate in place.
- Other prospects.

Then a less brutish variation of Figure 6.6 is given in Figure 6.9. These are not formally identical, but we conjecture that the latter misses no useful optimizations returned by the former.

In a practical setting, it remains only to pick the "most efficient" optimization from those generated by Figure 6.9, but several difficulties remain. For one, choosing the one with best absolute run time is an undecidable problem. Stated briefly, if we could decide whether some arbitrarily expensive copy operation is actually evaluated when a compiled **svexp** program runs, then we could also decide whether an arbitrary Turing machine halts. Moreover, static efficiency measures are not helpful: sharing is a generalization of register allocation problems that are known to be NP-hard [GJ79]. That is, even our improved algorithm may output exponentially many (in the length of f) optimizations of f, and we will generally have to check exponentially many of them to find the one with the best static efficiency measure (unless $P = NP$).

How can we tame this poor worst case behavior? A promising course is to preprocess P_f, removing all but one prospect due to each $=$ and function call in f. For example, in (6.25), we choose to share r with either x or y, but not both. This treatment is in fact an implicit assumption of previous work in this area [GH89] and [Blo89]. While it still admits an exponential number of optimizations in the general case, we have empirical evidence that for the special case of procedures like that of Example 6.1, where all *let*-defined variables can be shared with parameter or return arrays using the prospects of P_f preprocessed as above, the order in which prospects are considered is irrelevant.[6]

Figure 6.10 exploits this observation by considering only a single permutation of prospects. Moreover, for procedures where total sharing as in Example 6.1 is not possible, we obtain good results with a greedy heuristic: label each prospect with a static estimate of payoff for the sharing it represents. Consider prospects (in sets U and V of Figure 6.10) in descending order of payoff.

With reasonable assumptions, this algorithm runs in polynomial time. If the

[6] A proof would be extremely unwieldy with the current notations and algorithms. Better expository and proof methods are a topic of future research.

Inputs: $\mathbf{proc_2}$ procedure f and sharing prospect algorithm P.

Output: Set of all sharing optimizations of f possible with P.

Method: Let $U = P_f^-$ and $f' = f$
while $U \neq \emptyset$
  remove the first prospect $[b/a]$ from U
  if Figure 6.4 returns *correct* on f', a, b
    replace U by $U\,[b/a]$
    replace f' by $f'\,[b/a]$
if f' operates in place, i.e. $f'(\ldots, a, \ldots, [\xi_1]a[\xi_2]) \rhd \cdot$ then
  for all respective argument–result pairs t, s of recursive calls to f
    replace f' by $f'\,[[\xi_1]t[\xi_2]/s]$.
let V be the set of remaining prospects $[b/a]$ in P_f
while $V \neq \emptyset$
  remove the first prospect $[b/a]$ from V
  if Figure 6.4 returns *correct* on f', a, b
    replace V by $V\,[b/a]$
    replace f' by $f'\,[b/a]$
output f'

Figure 6.10: Final algorithm returning all optimizations of f given P.

length of f is n, then it considers $O(n)$ prospects. The algorithm of Figure 6.4 checks all pairs of references to variables in the prospect, hence its run time is $O(n^2)$. Assume that types are $O(n)$ in length. Let $C(\ell)$ be the cost of symbolic comparison of two dimension expressions of total length ℓ to establish S by the Disjoint Iteration Theorem (Theorem 6.9). Useful comparisons are possible in $O(n)$ time for the average case and $O(n \log n)$ time in the worst case. However, optimal comparison is NP complete: min and $+$ can simulate boolean *and* and *or* in a reduction from SAT. Thus the cost of comparing two references is dominated by the $C(n)$ comparison cost in any reasonable implementation. We conclude the total cost of the algorithm is $O(n^3) \cdot C(n)$. This time bound is discussed further in Chapter 7.

Chapter 7

Related Work and Conclusions

This chapter describes where the work presented thus far fits amid the work of others. It draws conclusions related to practical applications. Finally, it gives directions for future study.

7.1 Related work

The Alex paradigm is related to a broad range of computer language research.

7.1.1 Array semantics

The denotational semantics of arrays receives sparse treatment in the literature. Stoy [Sto77] mentions them in passing. Gordon [Gor79] gives a prototypical definition oriented toward recursively defined arrays. Though our notion of infinite tuples is common in logic, there apparently was no need for them in semantic domains until s-vals. Mou and Hudak [MH88] propose algebraic vector operators for expressing parallel divide and conquer algorithms, but these are defined only for the one-dimensional case. Compilation is not discussed.

7.1.2 Type inference

Our extension of [Mil78] to s-vals with map unification for axis types is strongly related to work in type systems for so-called object-oriented λ-calculi. In particular, [Wan87] uses record types called *rows*, which are maps from *labels* to types. These superficially resemble our maps from axis numbers to dimensions. Wand also generalizes unification along the same lines as our map unification. Two essential differences are that the order of pairs in rows is significant, and he has no "erasing" value for a field in a record like our $\circ$. None the less, Wand's exposition of complete type inference by reduction to most-general unification is the model for our presentation of axis inference. Unfortunately, Wand makes an error in proving his unifiers most general [Wan88]; the system is incomplete.

Concurrently, [R89] adopts a similar approach to record typing, including a $\circ$-equivalent value. However, this system omits row variables (our map variables). Rows consist of a fixed number of pairs, one for each of the set of labels that occurs in the entire program. The set must be gathered in a pass over the program preceding type inference.

Finally, [Wan89] expands on Rémy's work in two ways. First, he uses it to fix the error of [Wan87] and achieve a non-standard completeness result: He proves there are no principal types in his system, but that most-general types can be coded as a disjunction of type schema. Unfortunately, this disjunction can be of size exponential in the size of the program, even for rather innocent-looking terms, a problem our types do not share. Second, he reintroduces row variables under the name of *extension variables.* This leads to rows that closely resemble maps. He outlines an associated unification algorithm, different and more complicated than ours, whose correctness and efficiency are not examined. We conjecture that our technique for map unification is adaptable to this case.

None of this work involves extents or dimension inference in a second pass. Instead of dimensions, record fields have types of their own, which are inferred in the same pass with labels.

7.1.3 Compilation

MIMD vector operations. The work of [CBF91] deals with generation of efficient vector codes for MIMD processors using, as input, functional data flow graphs with vector tokens and vector operators as nodes. Part of the optimizer is an algorithm for deducing vector lengths from what is known about primitives. It is aimed at determining when the loops internal to primitives have matching bounds and non-interfering access patterns so they can be *fused* by the compiler into a single loop for efficiency.

APL. There is a large, old literature on compiling the array language APL that confronts some of the same issues as we do in compiling **svexp**. The state of the art is characterized in [Bud88]. In particular, the compiler discussed there uses a generalization of dope vectors similar to ours, and the overall goals of code generation are similar, despite the fact that APL has assignment and is dynamically typed. Budd points out that APL is a prime candidate for generating good vector code, and [CX87] bears this out. This bodes well for our eventual ability to generate good parallel code from **svexp**.

Conversely, code accumulations could be put to good use in compiling APL primitives. For instance, Budd encounters much the same problem we introduced in Example 5.2 while compiling the APL catenation primitive, but adopts a less efficient solution because his compiler cannot rewrite code it has already generated.

Single assignment languages. Single assignment languages similar in principle to **proc** have been proposed as source languages. The most widely known is

SISAL [McG85]. The (mostly) functional language Id [Nik88] has similar qualities, though it was designed specifically for research in dataflow machine architectures.

Compile-time garbage collection. The general problem of copy elimination in functional languages subsumes the array copying in **proc** and has been studied since at least 1977 [Bar77]. Of modern treatments, [JM89] is representative in its use of *abstract interpretation* [CC77] for interprocedural analysis. In [JM89] it is used to predict when list cells can be safely reused as a program runs in order to avoid calling an allocator and associated garbage collector.

Abstract interpretation is a broadly applicable method based on evaluating the source program with respect to so-called *nonstandard semantics* over a finite, partially ordered domain. The domain abstracts values in the standard denotational semantics. Its properties ensure that the least fixed point of a recursive function abstraction can be computed exactly and in finite time by iterative approximation. Thus the "recursive knot" we referred to in several contexts is unraveled in a very general way. Proofs of correctness for algorithms based on abstract interpretation are so painless that they are seldom given in the literature.

Abstract interpretation's downfall is run time potentially exponential in the number of arguments of functions. Moreover, poor run time behavior tends to arise in practice. The problem of deciding whether a function parameter is certainly not used in the course of call-by-name evaluation (i.e. *strictness analysis*) is inherently simpler than compile-time garbage collection. Yet it was found too costly to solve with an abstract interpreter for real programs in [You88]. To be fair, this analysis included limited treatment of higher order functions; hence, more experiments are needed to understand average performance for purely first–order languages like **svexp**, but the outlook is grim.

Aggregate updates. We gave a brief history of the aggregate update problem in Section 1.3 and cited [HB85] and [Blo89]. They concentrate on the comparatively narrow problem of deciding when Hudak's **updt** operator can be compiled as a scalar assignment, which is conceptually similar to compile-time garbage collection, though the payoff of a successful algorithm is much greater. In contrast, [GH89] treats the more general problem of in–place function evaluation. Since all these are based on abstract interpretation, however, they share its potential performance problems. Our final algorithm of Figure 6.10 appears to subsume the capabilities of [GH89] (when generalized to handle sets of mutually recursive functions), which makes the otherwise discouraging $O(n^3)C(n)$ performance described in Chapter 6 a step forward.

7.2 Alex

It is appropriate here to describe in broad terms the Alex pictures that are the basis for **svexp**.

7.2.1 svexp subterms as pictures

An Alex window on the graphic screen represents a function definition. Boxes in Alex windows are s-vals. A fresh value is created by "picking up" an s-val box from a palette of tools with the mouse (i.e. clicking the mouse over the tool) and dropping it (i.e. releasing the mouse button) in a function window. Dropping a value close to the top or bottom of the window causes it to "stick" to the edge, making it a parameter or return value respectively. The user makes two values identical by clicking in the box of one and releasing in the other; on release, both boxes assume the same randomly chosen (but editable) color to show the connection. For instance, the identity function results from creating a parameter and return at the top and bottom of a fresh Alex window, then identifying them in this manner.

Primitives are applied by selecting other tools from the palette. For instance, the

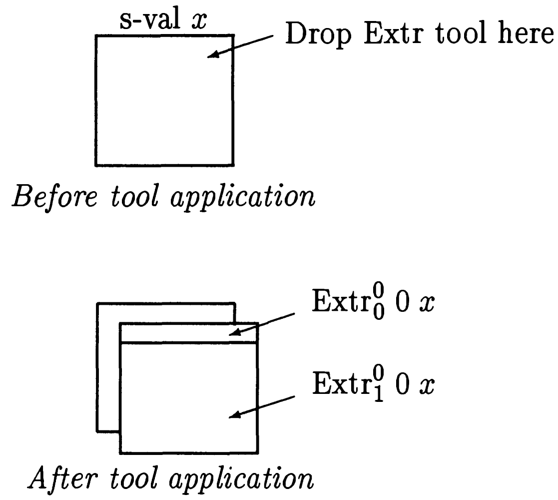
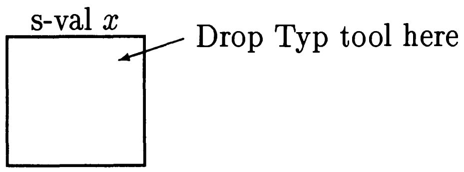


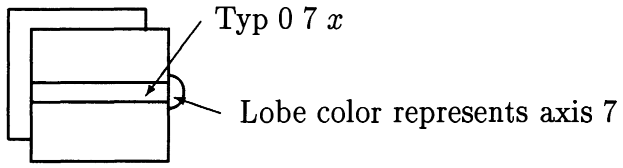
Figure 7.1: Typical element operations in Alex.

user applies the Extr tool to an s-val by picking it up and dropping it along a matrix edge to be "cut" by the operation as shown in Figure 7.1. The Typ operation shown in Figure 7.2 is similar, except it adds a colored lobe that stands for the outer axis of the typical components. Another Typ operation can be applied with respect to the same outer axis by selecting the lobe and dropping it along the appropriate edge of the argument s-val. Thus Typ 0 7 Typ 1 7 x looks like Figure 7.3.

User function calls are represented by boxes containing the name of the called function with rows of small "knobs" along the top and bottom to denote argument and return values respectively. One creates a new function call by clicking on the window of the definition (including the window where the call is wanted if it is to be recursive) and dropping it in the desired place. Alternately, there is a library menu for functions previously defined and saved; the user can pick up one of these as well. Function call argument values are defined by identifying the respective knob with a parameter or the result value of some previous computation as describe above. Outputs are put to use similarly.

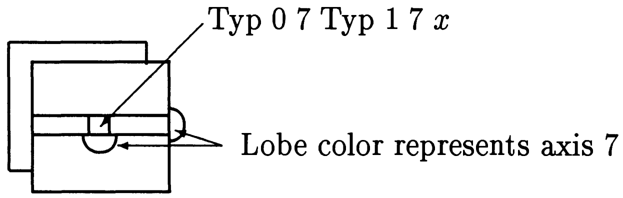


Before tool application



After tool application

Figure 7.2: Extract operations in Alex.



After second tool application

Figure 7.3: Multiple Typical element operations.

Conditionals are subwindows in the function window. A subwindow depicts either the true or false branch of the conditional at any one time; the user can toggle between them by clicking on the predicate knob at the top of the subwindow. This knob must also be identified with the desired boolean conditional value. There are output knobs on the subwindow's bottom edge to ensure that both branches define the same values. Subwindows may be nested to arbitrary depth to denote nested conditionals. Drawing constraints (e.g. keeping the cursor inside the conditional subwindow while copying a value defined therein) enforce scopes of conditional branches.

Type inference is an integral part of interactions. The data structure underlying the screen graphics is essentially a forest of abstract syntax tree fragments. When the function definition is complete, it forms a single tree. The type inference algorithm assigns a fresh type variables for the type of each value box not yet identified with a value of known type, then infers the type of each tree fragment. As long as unification succeeds, the graphics represent a potentially type-correct program. Interactions that cause inference to fail are refused. The vertical and horizontal screen axes represent inner s-val axes 0 and 1 respectively so that the convention of column and row vectors used in this text carry through to the appearance of values on the screen. The *hyperboxes* of [AC91] are useful for s-vals with more inner axes. The inference algorithm automatically yields the correct shape for the value of a function output once all its inputs are defined. Until then, it gets a rounded shape.

7.3 Generalizations

We invented **svexp** to explore s-vals and typical component operations and the parallelism they so naturally express. We have built a prototypical compiler that generates sequential C code from **svexp** to test these ideas. The early results are encouraging, but of course **svexp** in its current form is not a generally useful language. Thus we enumerate some logical directions to make **svexp** (and Alex) useful for other than data–parallel problems.

Computed partitions. Whatever the advantages of s-vals for expressing data–parallelism, a general purpose matrix language needs a conventional indexing operator and Hudak's **updt** primitive discussed in Chapter 1 for many practical problems; [ANP89] discusses the computation of histograms. Happily, we can incorporate both these into **svexp** with simple additions. I.e. define

$$\begin{array}{rcl}\mathcal{I}[\![\mathrm{Part}_1\ c]\!]V_1V_2 & \equiv & \text{when } V_1^{A_1,\alpha_1;C_1,\gamma_1} \text{ has } c \in C_1\\ & & \text{it's when } V_2^{A_2,\alpha_2;C_2,\gamma_2} \text{ has } C_2 = \emptyset\\ & & \text{it's } \lambda ij\,.\,V_1\,i\,j\,[(j\ c) + (V_2\,i\,\mathbf{0})/c]\end{array}$$

Thus Part_1 accepts two parameters—a general s-val and an inner-scalar that serves, roughly speaking, as an index. Part_1 replaces each inner matrix of its argument V_1 with a submatrix that is the same except the first n components along axis c are missing, where n is the respective inner scalar value of V_2. We obtain an s-val version of matrix indexing by composing Part_1 with Extr_0^0:

$$\mathrm{Extr}_0^0\ c\ \mathrm{Part}_1\ c\ x\ i$$

where x and i are s-vals. In fact, this operator offers more than simple array indexing. If p is an inner permutation vector, then the following computes the respective permutation of each inner column vector of x:

$$\mathrm{Proj}\ 7\ 0\ \mathrm{Extr}_0^0\ c\ \mathrm{Part}_1\ c\ x\ (\mathrm{Typ}\ 0\ 7\ p)$$

If we also define the symmetric variant Part_0 of Part_1:

$$\begin{array}{rcl}\mathcal{I}[\![\mathrm{Part}_0\ c]\!]V_1V_2 & \equiv & \text{when } V_1^{A_1,\alpha_1;C_1,\gamma_1} \text{ has } c \in C_1\\ & & \text{it's when } V_2^{A_2,\alpha_2;C_2,\gamma_2} \text{ has } C_2 = \emptyset\\ & & \text{it's } \lambda ij\,.\,(j\ c) < (V_2\,i\,\mathbf{0}) \rightarrow (V_1\,i\,j),\ \mathit{rangerr}\end{array}$$

then we can also fabricate an s-val version of **updt**. For e.g. column vectors:

$$\begin{array}{l}\mathbf{updt}\ v\ i\ x \stackrel{\mathit{def}}{=} \mathrm{Join}\ 0\ (\mathrm{Part}_0\ 0\ v\ i)\\ \qquad\qquad\qquad\qquad \mathrm{Join}\ 0\ x\\ \qquad\qquad\qquad\qquad\qquad (\mathrm{Extr}_1^0\ 0\ \mathrm{Part}_1\ 0\ v\ i)\end{array}$$

The compiler and optimizer accommodate these new operations as natural extensions; no new principles are needed.

Computed dimensions. We showed in Section 3.4.6 with (3.29) how to achieve s-val closure even if conditionals are allowed to produce different-sized outputs on their branches. This policy makes types as we have defined them undecidable. This has an important consequence: we cannot in general predict the size of a user function call result based on the size of inputs at compile time. Hence the procedure caller cannot allocate space for the result array. It remains to be seen how well the top–down compilation scheme using code accumulations of Chapter 5 and the optimizer of Chapter 6 can be adapted to this environment.

Higher order functions. Introducing s-vals into e.g. a general typed lambda calculus with function values opens many problems. For instance, the semantics of s-vals whose inner matrices contain functions (from e.g. s-vals to s-vals) require careful consideration. S-vals may of course be bound in the environments of function values with indefinite lifetimes during program execution. Thus the stack allocation of **svexp** is inadequate; a heap and garbage collector are required. This expense makes in–place evaluation all the more desirable, but similar analyses of higher order functions using abstract interpretation have proven intractable in theory as well as practice. Clearly, much work remains before higher order s-vals can be useful.

7.4 Summary

We have given some formal underpinnings for the Alex paradigm for representing matrix functions and have shown that Alex functions can be compiled to efficient imperative codes.

We began with an unconventional definition of matrices, upon which we built denotational semantics for the Alex-representative calculus **svexp**. We showed that **svexp** semantics are closed for s-vals, the matrix values that **svexp** primitives transform.

We described extents, the shapes and sizes of s-vals, and gave syntactic types to represent them. We constructed a sound and complete inference algorithm for extent types. For each subterm of a **svexp** program, the algorithm determines an extent type that describes all possible shapes and sizes the subterm can take at run time. Size constraints may be undecidable at compile time, but can be enforced by efficient run time checks in the compiled code.

We showed that the standard method for developing code generators is not ideal for **svexp** and proposed code accumulations as a way of letting the code generator transform code it has already emitted to avoid unnecessary storage allocation and copying in the compiled program. The result is compiled code that allocates storage only for only one inner matrix of each user function call argument and result.

We found that the last obstacle to performance lay in deciding when a compiled function could be safely called with a subarray of one of the parameters serving also to hold the return array or *vice versa*. We used pairwise syntactic analysis of references to determine when this was permissible, and we developed an algorithm to find a small set of good prospects for such sharing. Overall run time is $O(n^3)C(n)$ under reasonable assumptions, where n is the input length and $C(\ell)$ is the cost of comparing two symbolic dimension arithmetic expressions of total length ℓ.

Bibliography

[AC91] Bowen Alpern and Larry Carter. The hyperbox. Unpublished manuscript, April 1991.

[ANP89] Arvind, R. Nikhil, and K. Pingali. I-structures: Data structures for parallel computing. *ACM Transactions on Programming Languages and Systems*, 11(4), October 1989.

[App87] Andrew W. Appel. Garbage collection can be faster than stack allocation. *Information Processing Letters*, 25(5):275–279, 1987.

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, MA, 1986.

[AT89] Andrew W. Appel and Jim Trevor. Continuation-passing, closure-passing style. In *Proceedings of the 16th ACM Symposium on Principles of Programming Languages*, pages 293–302, January 1989.

[Bac78] John Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, August 1978.

[Bar77] J. M. Barth. Shifting garbage collection overhead to compile time. *Communications of the ACM*, 20(7), July 1977.

[Blo89] Adrienne Bloss. Update analysis and the efficient implementation of functional aggregates. In *Proceedings of the 4th International Conference on Functional Programming Languages and Computer Architecture*, pages 26–38, September 1989.

[Blo90] Bard Bloom. Course notes, CS611, Advanced programming languages. Department of Computer Science, Cornell University, November 1990.

[Bud88] Timothy Budd. *An APL Compiler.* Springer-Verlag, New York, 1988.

[CBF91] Siddhartha Chatterjee, Guy E. Blelloch, and Allan L. Fisher. Size and access inference for data-parallel programs. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 130–144, June 1991.

[CC77] P. Cousout and R. Cousout. Abstract Interpretation: A unified lattice model for static analysis of programs by construction or approximations of fixpoints. In *Proceedings of the 4rd ACM Symposium on Principles of Programming Languages*, pages 238–252, January 1977.

[CX87] Wai-Mee Ching and Andrew Xu. A vector code back end of the APL370 compiler on IBM 3090. *ACM APL Quote Quad*, 18(2):69–76, December 1987.

[FH88] Anthony J. Field and Peter G. Harrison. *Functional Programming*. Addison-Wesley, Reading, MA, 1988.

[GH89] K. Gopinath and John L. Hennessy. Copy elimination in functional languages. In *Proceedings of the 16th ACM Symposium on Principles of Programming Languages*, pages 303–314, January 1989.

[GJ79] Michael R. Garey and David. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness*. W. H. Freeman & Company, New York, NY, 1979.

[Gop88] K. Gopinath. *Copy Elimination in Single Assignment Languages*. Ph.D. dissertation, Stanford University, March 1988.

[Gor79] Michael J. C. Gordon. *The Denotational Description of Programming Languages*. Springer-Verlag, 1979.

[GVL83] Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, Baltimore, MD, first edition, 1983.

[HB85] Paul Hudak and Adrienne Bloss. The aggregate update problem in functional programming systems. In *Proceedings of the 12th ACM Symposium on Principles of Programming Languages*, pages 300–314, January 1985.

[JM89] Simon B. Jones and Daniel Le Métayer. Compile-time garbage collection by sharing analysis. In *Proceedings of the 4th International Conference on Functional Programming Languages and Computer Architecture*, pages 54–74. Addison-Wesley, September 1989.

[KR78] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice-Hall, Englewood Cliffs, NJ, 1978.

[KTC+87] Dexter Kozen, Tim Teitelbaum, W. Chen, J. Field, W. Pugh, and B. Vander Zanden. Alex—An alexical Programming Language. In *1987 Workshop on Visual Languages*, pages 315–329, Linkoping, Sweden, August 1987. Also in *Visual Languages and Applications,* ed. Ichikawa, Jungert, and Korfhage, Plenum Press, 1990, 147–158.

[McG85] J. McGraw, *et. al.* SISAL: Streams and iteration in a single-assignment language, language reference manual. Technical Report M-146, Rev. 1, Unversity of California, Lawrence Livermore National Laboratory, March 1985.

[Mey88] Alan Meyer. An efficient implementation of LU decomposition in C. *Advances in Engineering Software*, 10(3):123–130, 1988.

[MH88] Zhijing G. Mou and Paul Hudak. An algebraic model for divide-and-conquer and its parallelism. *Journal of Supercomputing*, 2:257–278, 1988.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* The MIT Press, Cambridge, MA, 1990.

[Nik88] Rishiyur S. Nikhil. Id (version 88.0) reference manual. Technical Report CSG Memo 284, M.I.T. Laboratory for Computer Science, March 1988.

[PJ87] Simon L. Peyton-Jones. *The Implementation of Functional Programming Languages.* Prentice Hall, 1987.

[R89] Didier Rémy. Typechecking records and variants in a natural extension of ML. In *Proceedings of the 16th ACM Symposium on Principles of Programming Languages*, pages 77–87, January 1989.

[Rea89] Chris Reade. *Elements of Functional Programming.* Addison-Wesley, Reading, MA, 1989.

[Rob65] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12:23–41, 1965.

[SS71] D. Scott and C. Strachey. Towards a mathematical semantics for computer languages. In *Symposium on Computers and Automata*, volume 21. Polytechnical Institute of Brooklyn, 1971. Microwave Research Institute Symposia Series.

[Sto77] J. E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*, volume 1 of *MIT Press Series in Computer Science.* The MIT Press, Cambridge, MA, 1977.

[Ung84] David Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. *ACM SIGPLAN Notices*, 19(5):157–167, 1984.

[Wan87] Mitchell Wand. Complete type inference for simple objects. In *Proceedings of the Second Annual IEEE Symposium on Logic in Computer Science*, pages 37–44, June 1987.

[Wan88] Mitchell Wand. Corrigendum: Complete type inference for simple objects. In *Proceedings of the Third Annual IEEE Symposium on Logic in Computer Science*, June 1988.

[Wan89] Mitchell Wand. Type inference for record concatenation and multiple inheritance. In *Proceedings of the Fourth Annual IEEE Symposium on Logic in Computer Science*, pages 92–97, June 1989.

[You88] Jonathan Young. *Theory and Practice of Semantics-Directed Compiling for Functional Programming Languages.* Ph.D. dissertation, Yale University, 1988.