Expressive Reversible Language: Aspects of Semantics and Implementation

Angel Robert Lynas

February 2011 PhD

Abstract

In this thesis we investigate some of the issues involved in creating a reversible variant of the formal software development language B. We consider the effects of regarding computation as a potentially reversible process, yielding a number of new programming structures which we integrate into an implementation-level language RB0, a more expressive variant of B0, the current implementation-level language for B.

Since reversibility simplifies garbage collection, in RB0 we make use of more abstract, set-based data types, normally available in B only at the specification level. Similarly, we propose extending the domain of abstract functions currently specifiable in B to allow them to become concrete functions, thereby furnishing B with a functional sub-language. We also investigate expanding the use of Lambda calculus from the abstract stage of B to the implementation. Unlike B0, RB0 will not disallow non-determinism, and can also specify what we call Prospective Value computations (which are described). The executable language implements all of these features.

After introducing some preliminary concepts, we review the work leading to the rise of Reversible Computing as a possible answer to the growing problem of energy dissipation in modern processors. We describe the language RB0, and demonstrate the use of its features, introducing the companion language RB1 and its role in the process. We then introduce our execution platform, the Reversible Virtual Machine (RVM), and translate some of the examples developed earlier into RVM code.

For the concrete functions, we provide a proposed syntax and translation schema to enable consistent translation to RVM, and introduce a postfix Lambda notation to link the RB0 specification to the RVM's own postfix notation. We provide comprehensive translation schemas for those parts of RB0 which would be found in B operations; these will form the basis of an automated translation engine. In addition, we look at a denotational semantics for Bunch theory, which has proved useful in formalising the underlying concepts.

Acknowledgments

I would like to express gratitude to my supervisor, Bill Stoddart, for inspiring the project in the first place and convincing me that I could do it. Thereafter for his seemingly endless patience and optimistic encouragement during times of halting progress, and not least for organising financial support where possible during the latter stages of the research.

Also thanks are due to Steve Dunne for being a supportive second supervisor, and to both for their Formal Aspects of Computing module, without which I would have struggled with many concepts.

Thanks to the European Social Fund for initial financial help, and the University of Teesside itself for various forms of financial assistance: the School of Computing for Teaching Assistant work during the first few years; and thanks to the Dean of Research for the School, Marc Cavazza, for providing a couple of bursaries where I was not able to fund myself, in particular during the thesis-writing stage (including an extension for illness). More generally, I am grateful that the University gave me the chance to undertake Higher Education twenty-odd years after I should have done.

Finally, a word of thanks to the Open University, for making available a way of substantially improving my general mathematical knowledge; indeed I hope to achieve a second degree (in Mathematics) in 2011, owing to their provision of effective distance learning.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this thesis we investigate a reversible programming language and a reversible formal language; we attempt to discern how expressive such languages can be. We also examine their suitability for use in an integrated formal program development environment; using B as our starting point, we consider changes to its implementation-level language such that the expressiveness of this is a good match for the executable target language, and the task of translation is eased.

1.1 Overview

Formal Programming Languages such as B must trace a path from specification to implementation, which imposes a number of limitations. Initially a mathematically tractable form of expression is used, which will describe the effects of a program without detailing the procedures; this stage is also amenable to proof — necessarily automated for a program of any size — that the specification does what it purports to.

At this point, however, the language used will not bear any relation to any mainstream executable language, nor will it be directly executable. The process of converting an initial specification to a form which can be translated to a suitable target language is "refinement"; a multi-stage process which will usually change the forms of data structures, and recast functions and operations to forms increasingly approaching those available in the target executable language.

At this final implementation level, then, the formal language will be simplified and relatively inexpressive compared to procedural languages such as C++, Java, or Ada, or more functionally-oriented languages, for instance Lisp, Scheme, or Haskell.

A number of useful features will disappear, as they have no analogue in the target language, and are not implementable. The implementation language is mainly geared to smooth translations. Conversely, there are likely to be capabilities in the target language which are simply not expressible at this implementation level, or perhaps even in the specification if a mathematical definition is not available.

So a part of our endeavour here is to allow certain features of the specification language to persist to the implementation, and to enrich the formal language by introducing certain other features which also have executable analogues. This second set of features relate to reversibility and its applications, e.g. backtracking, which informs another part of our endeavour.

As formal definitions and semantics have been (and are being) developed with regard to reversibility, it is now in a position to be incorporated in the formal language in the form of various programming structures. These have been implemented in the Reversible Virtual Machine (RVM), a collaboratively developed project at Teesside. It forms an interactive programming environment in its own right, but its suitability as a target language stems not only from its implementation of virtual reversibility, but its support for abstract mathematical structures based on set theory, and support for some aspects of functional programming, including anonymous functions. The implementation-level formal language can thus retain a number of desirable and expressive features which can now translate directly to an executable form.

A motivation for the interest in reversibility itself comes from the body of work on reversible computing begun by Rolf Landauer at IBM Research in the early 1960s [15]; this was inspired by problems in classical thermodynamics, for instance Maxwell's demon, and some informal results of John von Neumann. Landauer showed that the minimum power requirements of computations are related to the erasure of information — that is, to non-reversible steps. Subsequent work by Charles Bennett [3, 4, 5] (also at IBM Research) showed how computations of arbitrary complexity could be organised in a reversible manner. More recently, Zuliani [34] has formulated reversibility within a probabilistic guarded command language, relating it to the formal development process. This body of work shows that treating computations as essentially reversible processes runs with the grain of the underlying physical processes, and that reversible computing can be described in a formal way.

Another motivation is that we employ some applications of reversibility with regard to programming. An example is backtracking, which is available by various strategies in most languages, but will generally require some special setting up. The programmer must take explicit measures to restore previous values, to enable exploration of another branch of the state space. Our formulation exploits the interaction of infeasibility and non-deterministic choice to provide automatic backtracking with guarded choice constructs. These are simpler to program and amenable to formal analysis. Another example is a method to run computations provisionally, and gather a set of results, while not having any lasting side-effects on the state (other than the variable storing this set of results).

A third motivation is that reversibility can be used to simplify problems of garbage collection [2], making it practical to implement forms of data structure that are optimised for mathematical description, namely those based on (finite) sets; these include relations, functions, and sequences. The computational use of these requires temporary structures, which must remain in memory to enable returning to a previous state; reversing frees the memory used.

1.2 Thesis

Our thesis is that a formal programming language can remain highly expressive throughout its development process, from a mathematically-oriented specification which is suitable for automated proving techniques, to a procedural (though still mathematically tractable) implementation-level language which can be translated to our target executable language. Moreover, this language can incorporate features such as automatic backtracking, abstract data structures based on set theory, provisional computations yielding a set of possible results without changing the state otherwise, and an expressive functional sub-language — capable of recursion, for instance — which can capture the results of computation without side-effects; both of the latter simplify the discharging of proof obligations. Although these features suggest a language abstracted away from the physical realities of computations, we argue that there are in fact links with the efficiency of computation conceived in the most fundamental way, that is by thermodynamic analysis.

1.3 Objectives

1. Survey Background of Reversible Computing, with respect to its motivation for our study; and consider its implications for programming and specification, together with the most appropriate way we can address these implications. Chapter 3 is devoted to satisfying this objective.

2. Show how reversible constructs can be expressed in formal terms and in corresponding programming language terms. Both the simple guard and choice model and its extension to Prospective Value computations will be included. This will also demonstrate the use of Bunch theory in semantic description. This objective is addressed in chapters 4 and 5.

3. Provide demonstrations and case studies of the use of these constructs, with analysis of their development from specification to implementable language. Chapters 4 and 5 address this objective.

4. Document changes and extensions to ANSI Forth to transform it to RVM Forth. This is the subject of the first part of chapter 5.

5. Propose an extension of the role of constants defining functions, so that as well as being available at the specification (abstract) level, such functions, coded in a suitable way, may also be present in implementations. The first part of chapter 6 addresses this objective.

6. Discuss expanding the domain of Lambda abstraction to the implementation stage, and describe a means of implementing lambda expressions as part of our execution platform. This objective is the subject of the latter half of chapter 6.

7. Provide Translation schemas for B operations, from RB0 to RVM. This is addressed in chapter 7.

8. Begin process of formalising Bunch theory by means of a denotational semantics linking it to Set theory, in partial answer to critics who claim that Bunch theory has no formal model, and object to its use. The contents of chapter 8 are concerned with this objective.

1.4 Chapter Overview

2. Preliminaries. We introduce certain preliminary areas, outlining the relevant features along with some limitations of the existing system, and a small demonstration program. We also describe the Generalised Substitution Language GSL, which forms the core of the extended language; Bunch Theory, which simplifies the presentation of our semantics; and we include a general description of Forth, upon which the execution platform for this extended language is built.

3. Reversible Computing. We review the background and the previous work done concerning the physical processes underlying computation, and the desirability of reversibility as a way of reducing energy dissipation. This will become increasingly important over the next few decades as the drive towards greater speed and miniaturisation continues. The connection between erasure of information and irreversibility is established, and we demonstrate a step-by-step reversible assignment statement; as memory updates form the core of any computation, this has far-reaching consequences.

4. Reversible B. We introduce and examine the changes required for the existing B implementation-level language (B0), and describe the new variant RB0 (Reversible B0), which is based on the GSL syntax underlying B0, with a number of restrictions lifted. We briefly discuss the programmer-oriented variant of RB0, RB1, which is under development. We introduce Prospective Value expressions, and the $S \diamond E$ construct, and describe the capabilities of this form. We also include two case studies, the larger of which is revisited in the next chapter, to demonstrate and explore the expressive capabilities of RB0.

5. An Implementation Platform for our Ideas. We describe the RVM, the novel version of Forth which is our experimental platform, showing the changes required to support virtual reversibility and abstract data types. This is mostly from a programming interface point of view, although the internal additions are briefly discussed. An example from chapter 2 is expanded to compare translation to RVM with translation to C, and a longer case study from chapter 4 is translated informally.

6. A More Powerful Implementation Language. We describe a functional sub-language of concrete constants, which are not currently allowed to persist to B0, explaining why they are useful from a formal proof point of view as well as a programming point of view, for instance their support for recursion, which is also implemented in the RVM. We provide some examples and a proposal for standardising their form in RB0 such that it could be subjected to machine translation. We also examine Lambda Calculus, on which much functional programming is based, and outline its current status in B, along with a way it could play a larger role, given that support for it has now been incorporated into the RVM. We introduce a postfix notation for lambda expressions to facilitate the translation process.

7. **Translation Schemas for B Operations.** We provide preliminary forms of the translation schemas for the eventual translation of RB1 into RVM; its current status takes the RB0 form as its basis, but includes corresponding equivalent versions in B's Abstract Machine Notation where appropriate. We also mention some areas in which the information contained in RB0 would not enable the most efficient translation into RVM, so this could perhaps be supplied in RB1. The latter will be a little more programmer-friendly, but also more RVM-friendly, providing hints for efficient compilation which can be ignored as far as formal analysis is concerned.

8. **Theoretical Underpinnings for Bunch Theory.** Bunch theory is based on Set theory, which is foundational in mathematics, but arises from regarding collection and packaging as independent concepts. The packaging concept is removed, and Bunch theory deals with the properties of the remaining collection. We explore a way of relating bunches to Set theory using denotational semantics; the established Set theory results can then be used to verify the axioms of Bunch theory.

9. **Conclusions and Further Work.** In this chapter we draw some conclusions and outline some possible directions for future work arising in the relevant areas.

1.5 Contributions

Overall, the thesis argues that it is possible to construct an expressive reversible language with a simple formal semantics.

We follow the approach to reversibility formulated by P Zuliani [34], using a history stack to preserve information that would otherwise be lost during execution. However, we improve on Zuliani's formulation of reversible assignment statements by providing analysis of reversible assignments in which the reverse operations are themselves reversible, this necessary detail having been omitted from Zuliani's formulation.

We make use of "Prospective Value" expressions $S \diamond E$ representing execution of S, evaluation of E, and reversal of S, undoing any side effects. These have previously been formulated and discussed in [31], and we extend this discussion by giving the referential transparency rules for such terms, which differ from those that can be used with conventional expressions.

We propose an expressive expression language, forming, in effect, a functional sub-language. This language includes lambda expressions and supports functions as first class objects. It also allows the encapsulation of state within functional programs through the use of prospective value expressions.

To implement the functional sub-language on the RVM we provide a postfix formulation of lambda notation, and we implement lambda notation and closures on the RVM. We provide local variables that can serve either as the conventional stack frame variables of a sequential programming language, or as the binding variables of a function expression, whose bound values are able to persist beyond the variables' syntactic scope.

We illustrate the programming techniques made available by our approach through a number of programming case studies which are considered both through the high-level language under development and directly using the postfix Forth language of the RVM.

We sketch the translation of high level language structures to the RVM by means of "translation schemas", borrowing notation from denotational semantics.

We make use of bunch theory throughout the thesis, and bearing in mind its innovative mathematical nature we take some mathematical steps towards justifying the soundness of its use by providing a denotational model for the part of the theory used.

Chapter 2

Preliminaries

2.1 The B Method

2.1.1 Rationale and Method

Formal Methods encompasses a number of ways to make programs reliable, that is to perform exactly according to specifications, *providing* the inputs are within specified constraints. For this level of confidence, the program specification must be amenable to being mathematically proved, to show that the outputs and effects are indeed those required. This is particularly vital in critical systems, usually embedded, where safety is paramount.

To this end, a mathematical model is constructed embodying the situations which the program will deal with. The techniques of discrete mathematics are generally used, and the core concept is **state**. The state of a program is defined as the value which each of its variables currently has. Certain relationships between these variables must be maintained whatever changes are made to the state, and the values will be constrained according to the type; the set of these conditions is the **Invariant** of the system.

Any operation, then, which changes this state must only change the variables it is allowed to change; moreover the Invariant must be preserved. In constructing the model, the toolkit of discrete mathematics will be used to encapsulate the real-world objects and their relationships. The mathematical objects are scalar values, symbols, sets, sequences, functions and relations — each of which may be constant or variable. Naturally, certain simplifying assumptions will be made in the construction of this model (but only justifiable ones).

2.1.2 Provability

An important concept in the constraint of the system to only certain configurations is that of **type**. Some types are built-in, for instance $\mathbb{N}$, that of Natural Numbers[1]. Often a type is defined by the specifier as one of a set of symbols, e.g. $\mathit{RESPONSE} = \{\mathit{yes}, \mathit{no}, \mathit{maybe}\}$. Now an input or output, or any variable, can be specified as

[1] $\mathbb{N}$ includes zero here; for $\mathbb{N} - \{0\}$ we use $\mathbb{N}_1$

being of type *RESPONSE*, and at all times it *must* take one of these values in the set. Type-checking to ensure this is one element of the program proof. Naturally types can be more complex: power sets, functions, relations, sequences, or subsets of a type such as $\mathbb{N}$.

Operations on the system may be simple in themselves, or more involved. In the latter case, these will generally be split up into smaller operations where possible, as these will be more amenable to proof (in particular automated proof). One limitation of B in this regard is the amount of constraints it has in place on the interaction of smaller operations, which can lead to unnecessary over-complication in the model. The reason for this is partly historical, as the provision of automated proof support was urgently sought for the launch of B, and the constraints minimised interaction of operations to a degree where this became possible. The existing system of constraints is one motivation, however, for expanding the capabilities of B, as the structure of a model is one of the ways in which it is expressed; we examine this in more detail later.

The B method is largely geared toward automatic proof, implemented within an integrated development environment which allows a complete development from initial specification through to implementation language, then finally a translation to executable code. The proving systems are equipped with a number of axioms regarding the mathematical structures and operations used in the specification, and a larger number of theorems. They also have rules of inference via which the assertions implicit in the specification can be checked against the theorem library.

Occasionally, extra levels of this capability will have to be called into play, using more sophisticated techniques, and sometimes user-supplied theorems are necessary when the system cannot make the logical connecting steps on its own.

The obligations incurred by the specification with regard to proof, aside from any operation being feasible and applicable in relation to the types used, mainly concern the preservation of the conditions in the Invariant. Even a simple program specification can generate many proof obligations, at varying levels of triviality. The obligations stem from the weakest precondition model introduced by Dijkstra [6] and refined by Abrial [1], and are algorithmically generated from the mathematical form of the specifications; we examine the proof model briefly in section 2.1.7.

2.1.3 Refinement

A specification is not a runnable program in itself, however. It must be refined, usually in various stages, with both the data structures and the operations on them becoming less abstract and non-deterministic, and closer to the more algorithmic implementation language. At each stage, the refinement must be linked with each previous stage, such that the proofs will still hold.

The final stage, still automated, will be translation to some form of code which can be compiled to an executable program. The translations from the algorithmic language of the final refinement (this is the implementation level) to code are governed by translation schemas, which should have been separately verified to provide an equivalent code to that of the specification.

The format of the implementation-level language is limited by the formats of the available target languages for code. This is one motivation for our proposal of another target language, RVM-Forth (chapter 5), which will allow a concomitant greater expressiveness in the implementation language, thus it need not be so far removed from the abstract specification language.

2.1.4 Abstract and Concrete Constants

B allows constants to be defined both abstractly and concretely, concepts related to the refinement process. Constants can be scalars or sets, whose instantiation can be deferred until the implementation phase (the concrete stage), or total functions.

The latter would not normally remain as functions until the implementation stage, unless implemented as arrays, for instance. We propose carrying some of the possibilities of specifying abstract constants, perhaps as recursive functions, into the concrete stage. Since the target language has facilities for both of these, translations could be achieved.

As the functions are "black box" operations, they have no direct effect on the state of the machine or its invariant, and thus incur no proof obligations in this respect. However, there are certainly *existence* obligations to consider, that is, does a function with the stated properties and expected outputs actually exist?

However a description of such a constant could go through a number of stages; from an abstract list of properties initially to a more concretely specified (perhaps recursive) function which could then be translated to RVM code. Here is where a richer language for concrete constants could be developed, to bridge the gap between the currently available listing of a function by its abstract properties in the **PROPERTIES** clause, and something that could be related directly (via a translation schema) to RVM code.

2.1.5 Generalised Substitution Language and Program Constructs

A small number of symbols and notational conventions from GSL, the Generalised Substitution Language [1], are used in this document to describe program constructs and as placeholders for B notation. We provide a brief introduction to the most important elements here; in the following, S and T are program statements or fragments (or complete programs), G and P are predicates, x is a variable of some type and A is a set of the same type as x. The main constructs are summarised in table 2.1.

The precondition will not concern us a great deal, as its being false does not prevent S from being executed. It simply means that the outcome of S guaranteed under P is no longer guaranteed. We will assume that any preconditions hold in our examples.

Name	**GSL**	**Example**	**Remarks**
Empty Operation	skip		
Assignment	$:=$	$x := 1$	RHS same type as x
Guard	$\Longrightarrow$	$G \Longrightarrow S$	S only executed if G is true.
Choice	$[]$	$S \, [] \, T$	Non-deterministic
Assignment from set	$:\in$	$x :\in A$	Any member of A
Unbounded choice	$@z.S$		Any variable z used in S
Sequential composition	$S \, ; \, T$		S followed by T
Precondition	$\mid$	$P \mid S$	

TABLE 2.1: Generalised Substitution Language

The behaviour of a program in a guard situation is important, however. We note that S is only executed if G is true; but in the case that G is not true, execution would normally move on to the next instruction, unless an explicit ELSE clause was supplied. So a simple IF-THEN construct,

IF G THEN S END

is expressible in more primitive concepts as

$$G \Longrightarrow S \, [] \, \neg G \Longrightarrow \text{skip} \tag{2.1}$$

We note that this formulation could be troublesome if implemented literally, if the guard G had any side effects. There could potentially be a double evaluation of the guard. We adopt the convention, however, that the evaluation of guards in B should not entail side effects. In fact, as we shall see in later chapters, if necessary we could use and alter state in the evaluation of a guard inside a prospective value construction, which would leave the required boolean value and then reverse, undoing any changes in state.

The decomposition in (2.1) above shows that there is an implicit ELSE with a skip instruction:

IF G THEN S ELSE skip END

Removing this implicit ELSE raises the question of what will happen if the condition is false; we now have the guarded command $G \Longrightarrow S$ standing alone, in a situation where progress becomes infeasible. One answer is to go into reverse, and we describe this option in detail in section 4.2, including the part the choice construct plays in where the reversing goes back to. This notation is also an example of Abstract Machine Notation (AMN), which is a sort of "syntactic sugar" used to make B machines more readable.

Non-determinism features in both choice and assignment from a set; one feature of B and its refinement structure is that non-determinism must not appear in the final implementation specification, however since our target language uses non-determinism directly, this limitation need no longer apply. Our reversible environment uses this in a number of ways, one particular formalism being described in section 4.4.

2.1.6 Structuring and Limitations

One aspect of structuring a B model is that an operation cannot be called within the same machine, the idea being to facilitate the discharge of proof obligations without having to work out the ramifications of nested operation calls. So, in a B specification, the functionality of larger models is parcelled out among several machines, linked in a hierarchy using various available kinds of visibility. These allow certain levels of access to variables, constants, the invariant, or operations of the specified machine (listed in an appropriately named clause).

Two basic modes are **USES** and **SEES**, which provide a machine with read-only access to the state of the machines mentioned in the clause. There are subtle differences between these modes, which are not central to our concerns.

Also there is the **INCLUDES** mode, wherein the state information is effectively separated out among machines, along with invariant-preserving proof obligations. Thus a machine's operations will preserve its own invariant; if these are called by an including machine, it is up to that machine to see that the operations preserve its own invariant.

A consequence which seems unnecessary is that even query operations cannot be called within the same machine; all these do is report on the state, without changing it, and thus never incur proof obligations. There is no way of flagging such operations as query-only at present; conceivably they could be collected in a different clause, separate from other operations.

2.1.7 Proof Model: Predicate Transformers

In the model due to Dijkstra and Abrial, the substitution S is viewed as a predicate transformer, written $[S]$. The transformation is on a predicate involving certain variables of the current state, and is intended to establish a *postcondition*, which is another predicate in which the variables may have been changed.

If Q is such a postcondition, we write $[S]Q$ for the weakest precondition that S will establish Q. There are rules for calculating the weakest preconditions for all substitutions, and we present a simple example. In the following, the notation $F\langle E/x\rangle$ refers to the expression F with all free occurrences of x replaced by the expression E.

The weakest precondition rule for an assignment $u := E$ is

$$[u := E]Q \;\hat{=}\; Q\langle E/u\rangle$$

So for assignment $x := 7$ and postcondition $x < y$, we have

$$\begin{aligned}[x := 7](x < y) &= (x < y)\langle 7/x\rangle \\ &= 7 < y \quad = \quad y > 7\end{aligned}$$

If the precondition that $y > 7$ holds before the assignment is executed, the postcondition $x < y$ will hold afterwards; this is all that is required, and is thus the weakest precondition.

The rule for a guarded substitution is

$$[G \Longrightarrow S]Q \ \hat{=} \ G \Rightarrow [S]Q$$

In this case, where G is false, the right hand side says that anything can be implied, certainly including $[S]Q$ (known as behaving "miraculously"). But as the guard is false, the substitution will not take place; without an explicit alternative guarded by another condition, progress becomes infeasible. This, as noted earlier, is the situation where reversibility is invoked.

The basic predicate transformer rules are shown in table 2.2.

Substitution	**[Sub]Q**
skip	Q
$u := E$	$Q\langle E/u\rangle$
$G \Longrightarrow S$	$G \Rightarrow [S]Q$
$S \, [] \, T$	$[S]Q \wedge [T]Q$
$x :\in A$	$\forall a.(a \in A \Rightarrow Q\langle a/x\rangle)$
$@z.S$	$\forall z \cdot [S]Q$
$S \, ; \, T$	$[S][T]Q$
$P \mid S$	$P \wedge [S]Q$

Table 2.2: Predicate Transformer Rules

The other aspect of substitutions which we make some use of is the before-after predicate prd (name coined by Abrial), which reconciles the predicate transformer interpretation with a relational view of substitutions. The latter relates the variables which can be affected by a substitution S (also known as the *frame* of S), forming them into maplets of before-values and potential after-values; that is, we have

$$\mathsf{rel}(S) = \{x, x' \mid \mathsf{prd}(S) \cdot x \mapsto x'\}$$

The before-values are written as the plain variable, while the after-values are primed. We define prd as

$$\mathsf{prd}(S) \ \hat{=} \ \neg[S]x \neq x'$$

In this, x can stand for any variable in the frame of S; it is also possible to write, for instance, s and s' to refer to the entire frame.

The double negation in this definition is a semantic subtlety which captures certain aspects of non-deterministic behaviour for relevant substitutions, whereas "cancelling out" the negations would fail to do this.

2.1.8 Proof Obligations and Code Generation

We introduce a small and slightly pathological machine with a single operation to examine the proof obligation mechanism of the B-Toolkit software (B-Core), and its code generation. The machine has no variables and initialisation, so no proof obligations will refer to these. Initially there would be an abstract specification, but in this case it is very similar to the implementation, so we list only the latter as follows[2]:

```
IMPLEMENTATION  simpleI
REFINES  simple
OPERATIONS
  $b, c \longleftarrow \mathbf{S}(a) \ \hat{=}$
    IF $a > 4$ THEN
      $b := a + 1;$
      $c := a + 2$
    ELSE
      $b := a + 1$
    END
```

The only differences from the abstract specification are that the latter has a precondition (that $a \in \mathbb{N}$) and that sequential operations are not allowed in specifications, so the two assignments are made in parallel.

The first line of the operation says that b and c are "output parameters" (return values) of the operation S with input a. The problem with the above is the question of what happens when $a \leq 4$, when the parameter c has no value assigned to it. As there is no initialisation of these parameters, its final value is undetermined in this case. If no assignment were made at all to c, even its type could not be determined. Here the previous clause of the IF statement provides the type information.

So generating proof obligations for this machine leaves one undischarged, with proof goal

$$a \leq 4 \Rightarrow c_0 = c$$

which expresses the uncertainty concerning the contents of c if it is not assigned to.

Code can still be generated from this, the B-Tool generating C code by default. Again simplifying the variable names for readability, it produces this for the operation S:

```
void S (int *b, int *c, int a) {
   if (a > 4) {
    *b = a+1;
    *c = b+1;
   }
   else {
```

[2]The B-Toolkit does not allow single-letter variables; but in the interests of readability we have simplified the listings in order to allow them (and eschewed some BEGIN-END keywords).

```
    *b = a+1;
   }
}
```

Since C functions only return one value, we cannot return more than one without some enclosing structure. So the generation uses call-by-reference to assign the outputs. Later we examine the code produced when a calling machine is specified that uses this kind of operation and others inside an operation of its own; the call must be made with the addresses of the variables corresponding to b and c.

As C does not initialise declared variables, in the event of an unassigned c being used in a subsequent operation, the result will be entirely unpredictable, as any random value could be in that location.

The RVM, as we shall see in chapter 5, can use the stack to pass several return parameters, so it need not follow this model (although it can).

2.2 Introduction to Bunch Theory

Bunches are an invention of Eric Hehner [13], which are intended to provide a simpler data structure than sets for use in Computer Science. A bunch is the "contents" of a set, without the packaging; therefore nested bunches are not possible.

Since its invention the theory has been developed [19, 33, 14] and bunches have proved very useful in research work.

To put our bunch work on a firmer footing formally, we describe a denotational semantics for the theory in chapter 8, which explicitly links Bunch Theory with established results in Set Theory.

2.2.1 Bunch Theory Notation

As bunches have no set-style packaging, we make no distinction between an element and a singleton bunch. A bunch does not *contain* its elements, it *consists of* its elements.

The empty bunch is **null**. Table 2.3 defines some basic notation and relationships, where A and B are bunches.

Bunch Expression	Definition
A, B	Union of A and B. Commutative.
$A, \mathbf{null} = A$	**null** is identity for union
$A \setminus B$	Elements of A not in B
$A{'}B$	Intersection of A and B
$A{'}\mathbf{null} = \mathbf{null}$	Absorption by **null** under intersection.
$A : B$	A is a sub-bunch of B

TABLE 2.3: Bunch Relationships

As brackets may be needed to clarify precedence of operations in bunch expressions, and the comma is an operator representing bunch union, we use the maplet symbol $\mapsto$ to create ordered pairs.

Where g is a predicate and E a bunch expression, we define the guarded bunch $g \longrightarrow E$ by the property

$$(g \Rightarrow (g \longrightarrow E = E)) \wedge (\neg g \Rightarrow (g \longrightarrow E = \mathbf{null}))$$

Bunch comprehension is introduced by the symbol §, and has the general form

$$\S x \bullet g \longrightarrow E$$

which defines the bunch of values taken by E as x ranges over values which satisfy g.

Function application we define at the level of elementary maplet application as follows, where a, b and c are elements:

$$\text{apply}(a \mapsto b, c) \ \ \widehat{=} \ \ (c = a) \longrightarrow b$$

Thus an application where $a \neq c$ will result in **null**. A function (or relation) is a set of maplets; the bunch operator for extracting the contents of a set is ~. So application to a set f generalises to

$$f(a) \ \ \widehat{=} \ \ \text{apply}(\sim f, a)$$

which will return an elementary bunch if f is a function, and a bunch if f is a relation, or of course **null** if a is not applicable. Both cases (function and relation) are covered by the same definition; and in terms of bunch comprehension we have

$$f(a) \ \ \widehat{=} \ \ \S b \mid a \mapsto b \in f$$

2.3 The Forth Language and Virtual Machine

Forth, a computer language which dates back to the 1960s, is the base language for the RVM. There are several varieties of Forth — by its nature it is highly customisable — but all have certain features in common.

A Forth program consists of “words” (strings of characters) separated by spaces; the words can perform simple operations or themselves be composed from many other words to perform more complex operations. The language is “interpreted” line-by-line in interactive mode (though new definitions are compiled); each word is executed as encountered as the input line is processed. There is therefore very little in the way of syntax in Forth, with the exception of constructs related to program flow (if-then, or loops), which may well be written in Forth in any case. The construction of such higher-level words furnishes the other examples of syntax.

Much of the time, however, any word can follow any other word; the limitation on this is the *stack*, the last-in first-out data structure on which Forth is built.

This holds integers, which can also represent memory addresses. Almost all Forth words have some effect on the stack, for instance removing the top two items, performing a computation, and putting the result back.

This limits the interactions of words in two ways: the stack must not "underflow", which occurs if a word expects an item there and finds none; also, although there is in general no type-checking built in, the values on the stack must be of the right type. In particular, as memory locations are treated as integers, an operation which expects a memory location to access would probably cause a segmentation fault if some random number were on the stack. The provision of type-checking has been explored [22, 24], but it is not a part of the current standard.

The most convenient way of using a stack computationally is to use Postfix, as the words then should have the items that they need already on the stack. There are some exceptions to this, when an input should not be evaluated as an instruction, but as a string of characters, for instance (the naming of a new operation is such an occasion).

An example of an interpreted interaction is

```
3 4 < .  (enter)  -1 ok
```

The Forth code `3 4 < .` is entered from a console. This pushes 3 and 4 onto the stack, the command $\boxed{\texttt{<}}$ is a test which removes them and, since $3 < 4$, pushes a true flag. The command $\boxed{\texttt{.}}$ removes and prints the top stack value as an integer.

Since Forth commands can be any character string, we use a box to clarify the display of otherwise ambiguous commands.

A new command is defined between the commands $\boxed{\texttt{:}}$ and $\boxed{\texttt{;}}$. The command name comes first, followed by a stack comment, then the body of the definition:

```
: >= ( n n -- f )
< NOT ; ok
```

The comment is a "stack signature", and specifies that two integers must be found on the stack, and that a flag will be pushed afterwards. The new command $\boxed{\texttt{>=}}$ is now accessible in the same way as any other:

```
3 4 >= .  (enter)  0 ok
4 3 >= .  (enter)  -1 ok
5 5 >= .  (enter)  -1 ok
```

Since the effects of Forth words are generally on inputs found on the stack, it is important that a stack signature is given, to show what is expected on the stack and what will be left at the end. There are certain conventions we will use for this, as regards which letter or symbol corresponds to which input type, which are listed in table 2.4.

Sometimes extra information is added to show the effect in more detail, for instance

Symbol	Type
`x`	Item of any type
`n`	Integer
`$`	String
`f`	Flag (boolean)
`x^, n^`	Address of an object or integer
`s`	General set.

Table 2.4: Stack Signature Elements

```
( s1:x.P s2:x.P -- s3:x.P,  where s3 = s1 union s2 )
```

In this case the sets `s1` and `s2` are numbered and have a structure specified (which need not be an elemental type) introduced with a colon. The output is a set of the same type, and the additional information shows that it is the union of `s1` with `s2`.

We examine the specification of sets in signatures in more detail when the set capabilities of the RVM are discussed in section 5.1.2, as both the types and signatures use postfix, and are not immediately intuitive.

2.4 Summary

In this chapter we have introduced the preliminary background for the Thesis. We covered the more relevant areas of B, which provides an established formal framework for the work, and GSL notation; Bunch Theory, and the basics of Forth.

In the next chapter we examine a major motivation behind this work, which underlies the idea that computation is a potentially reversible process, and the consequences of this idea.

Chapter 3

Reversible Computing

3.1 Physical Reversibility

The notion underlying the origins of the Reversible Computing endeavour is that *information is physical.* Reversible physical processes are not unusual, all the more so if idealised objects and conditions are envisaged (frictionless, isothermal environments, for instance). And information cannot exist in the absence of some physical substrate; whether organic (the activity and configuration of neurons), or inorganic, such as the many ways of storing information in computing.

We look first at this physical manipulation of information and how it relates to Reversible Computing. An initial motivation was how to reduce the energy lost as waste heat in computation; in the forties and fifties this was very high. Around this time, an idea arrived at by John von Neumann was that each elementary operation, or "act", of computing would dissipate $kT \ln 2$ joules. Here k is the Boltzmann constant (in SI units, about $1.38 \times 10^{-23}\,\mathrm{J\,K^{-1}}$), while T is the temperature of the environment in kelvins and $\ln$ is the natural logarithm. The reasoning behind this was that every such elementary act would remove one bit of uncertainty from the result, and so reduce the entropy of the system by a corresponding amount; we arrive at the same result in a different context below.

As a motivating example for examining reversibility, we can imagine a sort of billiard table, without pockets, but also without friction (thus in a vacuum). The idealised balls on this table are perfectly elastic, and as they move and rebound from the cushions and each other, they form a "conservative system", conserving energy. They also conserve the history of their movements; if they were stopped and the motion of each ball exactly reversed, they would unwind their earlier movements, visiting all the configurations they had been in along the way. However, if friction is re-introduced, the system is now damped and no longer conservative, and the balls will eventually come to rest, having lost all their kinetic energy to their environment. From this point there is no possibility of reversal, as the information of their previous movements has been lost. This example suggests a connection between damping (friction) and information loss, leading to non-reversibility.

We can begin a closer look at these concepts with a model of information storage of a very simple kind. Consider the cylinder in figure 3.1, which contains one molecule of gas; the behaviour of this averaged over time is that of an ideal gas. A zero is registered when the molecule is in the left half of the cylinder, achieved by pushing in the (frictionless) piston on the right. This would normally increase the temperature of the gas (i.e. the speed of the molecule), but we assume an *isothermal* compression: this takes place slowly enough for the increase in energy to be absorbed by the environment.

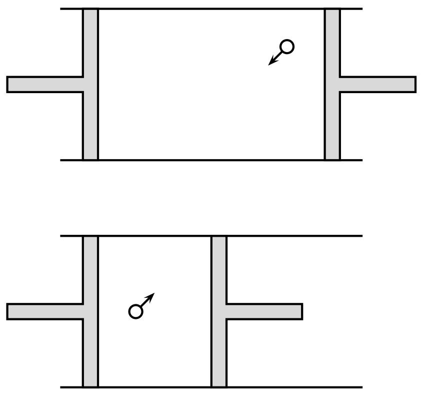

FIGURE 3.1: Molecule in a cylinder.

The Ideal Gas Law relates the pressure P and the volume V of a gas at temperature T by the formula $PV = NkT$, where N is the number of molecules. In the cylinder we have a length L between the pistons, which have an internal surface area of A. The volume occupied by the gas is therefore $V = AL$; for $N = 1$ the foregoing gives

$$PAL = kT.$$

The force exerted by the gas on one piston is PA, which is thus kT/L. Given an initial distance L_0, the minimum work needed to compress the gas to half its volume is given by the integration

$$\begin{aligned} W &= \int_{L_0}^{L_0/2} \frac{kT}{L}\, dL \\ &= kT \ln\left(\tfrac{L_0}{2}\right) - kT \ln L_0 \\ &= -kT \ln 2 \end{aligned}$$

This is the same quantity quoted earlier; note it does not depend on the size of the cylinder or the mass of the molecule. It is a general result when constraining a particle along one of its degrees of freedom, to express the change in entropy.

In 1961, Rolf Landauer [15] returned to von Neumann's statement, and undertook a more thorough analysis of the question; he showed that only irreversible operations have this inevitable dissipative cost. An irreversible operation is one in which the output does not uniquely determine the input — when the operation is executed, the input information cannot be reconstructed. So for instance an addition in $\mathbb{N}$ with result 7 could have had any set of inputs from the set $\{\{0,7\},\{1,6\},\{2,5\},\{3,4\}\}$.

A way of making such operations reversible is to store the inputs, or at least sufficient information to enable their reconstruction (in the case of addition, one need only store one input). The one operation that cannot be made reversible is that of erasure; by its very nature, information is discarded. So the only inevitable thermodynamic cost in computation arises from erasure.

Landauer considered the possibility of making computations reversible by storing intermediate results in such a way that the whole could be made reversible; this has an obvious drawback in that the available storage could easily be filled if a computation was non-terminating. Even disallowing such computations, we have at some point the obligation to re-initialise the storage, thus re-introducing erasure of information and its concomitant costs. But the notion of embedding a computation in a larger context which would store its history would prove fruitful.

A way of illustrating an irreversible act is using the visual metaphor of a "bistable well". This can represent numerous physical situations, e.g. electromagnetic configurations. We can use its metaphorical nature literally, as though it were a cross-section through some physical (but idealised) apparatus. In figure 3.2, the particle has stable positions in either well; its being in the left well represents zero and in the right 1. Gravity plays its usual role; there is no damping in the form of friction, so this is a conservative (energy-conserving) model, and time-reversible in the classical mechanical sense.

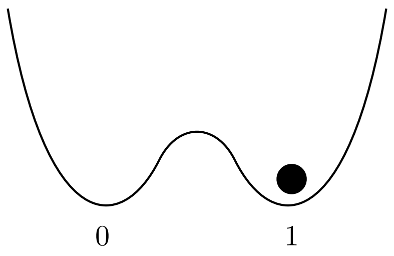

FIGURE 3.2: Bistable well model.

If the particle is in the 0 well, we can move it to 1 with no energy loss by "pushing" it up the slope (i.e. imparting energy to get it over the barrier), but recovering the energy from its fall on the other side; similarly from 1 to 0. But in each case, we would need to know the position of the particle first, as the direction of push and retardation differ for each one.

Supposing we do not know where the particle is, and are required to simply set it to 1 regardless of its starting point? This is a more likely candidate for an analogue of a computing operation, which would act in the same way on any data that it was presented with. However, in our conservative model, there is no single reset operation $\mathcal{R}$ that can do this and remain classically reversible, with a deterministic inverse $\mathcal{R}^{-1}$. Attempting to reverse from 1 after applying $\mathcal{R}$ is now non-deterministic, as the particle could have been in state 0 or state 1.

If a certain amount of damping is now allowed in the system, our single operation $\mathcal{R}$ could work; with a force applied slowly from the left, if the particle is in 0, it will be pushed over the barrier, and come to rest in the 1 well. If the particle was there all along, we would have the same result. But the damped system is now irreversible, as we have erased the history of the particle.

Landauer's view remained unchallenged until 1973, when Charles Bennett [3] proposed a way of rendering the recording of intermediate information as *also* reversible, via the example of a three-tape Turing Machine. The essence of this is to copy the intermediate data to a second blank tape, a history tape containing the "traces" of the forward computation. The latter might be viewed as a sequence of reversible operations — of transition functions from one state to another. In our terms, we might prefer to characterise them as reversible GSL substitutions.

The result of the computation is then copied to the third tape, and the program is put into reverse. Now the second tape provides inputs for the reversed program, using inverse transition functions which *uncompute* the history, rather than erasing or re-initialising it. But since the tape was blank beforehand, it is returned to this initial state by the reverse computation, while the original inputs are reconstructed. The result remains accessible on the third tape.

In our virtualisation of a reversible machine, the history tape becomes a history stack, an ideal data structure for its required operation. There is a one-off cost associated with initialisation; thereafter programs can be run reversibly without the wholesale and indiscriminate erasure of intermediate results.

We note that these early models operate in a world of discrete computing "jobs", with well-defined beginnings and endings, so a program with a particular task would run and terminate at some point with a definite result. Our examples elsewhere, indeed, are of this kind. The more recent paradigm is of continuously-running programs (the operating system itself, for instance, and many applications); it seems less obvious how the process is to be broken up.

Our earlier image of a frictionless billiard table was the inspiration behind *ballistic computing*, an idea of Fredkin and Toffoli [12], which proposed an implementation where the movement of the balls would be isomorphic to the desired computation. Versions of logic gates using the elastic collisions of these balls were designed, but cannot really be approached with a physical realisation. Even in theory, extremely small perturbations in the trajectory of one ball — caused by random thermal noise — would result in the rapid degeneration of the balls' movement itself to noise.

However, the principle remained as a guide for the concept of Conservative Logic [11], which proposed designs for more general logic gates with arbitrarily low power dissipation. The ballistic paradigm has been used more literally in recent research involving transistors using collisions between electrons, or ballistically controlled spin interactions [23].

In later work [4, 5], Bennett also proposed other models, most notably the Brownian model; this used reactions from biochemistry as an illustrative example, but the principle applies to any reversible chemical reaction. Typically, such reactions reach an equilibrium determined by the concentration of the major and minor reactants. Leaving aside any chemical details, we can represent the chemical configurations as states σ_1 and σ_2, and write the reaction as

$$\sigma_1 \rightleftharpoons \sigma_2$$

to denote a reversible reaction. Equilibrium is not static at the molecular level; the individual components continue reacting back and forth — moved probabilistically by thermal fluctuations, and the equilibrium is statistical at the macroscopic level.

A chain of such reactions can be imagined, with a "result" corresponding to the result of the computation represented by the states, thus

$$\sigma_1 \rightleftharpoons \sigma_2 \rightleftharpoons \ldots \rightleftharpoons \sigma_{n-1} \rightleftharpoons \sigma_n$$

In the chain, the reaction or current state can move forward or back with roughly equal probability, becoming a one-dimensional random walk. This will *eventually* reach the desired result state without energy expenditure, but this is in the limit of no reliable forward speed. It can be speeded up at the cost of some energy dissipation by introducing a bias, depending on the physical model used. This would give a greater probability of moving in the "right" direction at any given stage, although movement in both directions would still occur.

Bennett has also described a Brownian Turing machine [5] constructed from frictionless clockwork components, subject to small random movements which are able to move it from one state to another; with a certain amount of bias, this will also complete its computation without large energy loss.

More recently, pursuing commercially viable hardware, research is going on into Adiabatic Logic and circuitry [10]; as today's irreversible logic circuits approach their physical limits, these will provide a path to continue improving performance. At present, there may be a few decades before this becomes a commercial inevitability.

3.2 Logical Reversibility

A point that generally follows from consideration of physical reversibility is that a different programming paradigm is required, having the property of logical reversibility. This will filter down from some form of reversible specification language to a logically reversible implementation language. But there are different approaches to achieving the latter.

The more literal approach is to try and make *every* operation reversible. The Janus language developed by Tetsuo Yokoyama and Robert Glück [30] is designed from the ground up to be reversible, and to have this built into the syntax of every construct. A procedure can be called, or "uncalled" to run it in reverse. The designers show that local inversion is sufficient to provide deterministic inverse programs — so to transform a program into its inverse, recursive descent is used over the program structure to convert each procedure and its atomic components to their inverse forms.

This is also the approach adopted in a somewhat different way by Zuliani, in his paper on Logical Reversibility [34]. The mechanism is hidden from the programmer, and described in terms of pGCL (probabilistic Guarded Command Language, with some similar constructs to those in basic GSL). In terms of the basic GSL substitutions used in this thesis, we would require that every operation S had an inverse S^{-1}, such that

$$S\,;\ S^{-1} = \mathsf{skip}$$

However, most operations cannot be made automatically reversible as they are; a version must be created which would save its input to a history stack, avoiding erasure. Then a reversed operation has this available as input. For a simple assignment statement $v := e$ this might be achieved with

$$\begin{aligned} \text{Reversible operation } S_R &\ \widehat{=}\ \text{push } v\,;\ v := e \\ \text{Inverse operation } S_R^{-1} &\ \widehat{=}\ \text{pop } v \end{aligned}$$

Here, "pop v" is shorthand for "pop the stack and assign the value to v". The above satisfies

$$S_R\,;\ S_R^{-1} = \mathsf{skip}$$

but does not have the property of *stepwise* reversibility; for instance it contains the original irreversible assignment as a step, also pop v which is likewise non-reversible.

In table 3.1 we quote a few entries from Zuliani's original listing, adjusted for our GSL style. In this table, b is a boolean variable, which holds the truth value of the guard for the next choice in the reverse direction; again "pop b" is shorthand for "pop the stack and assign the value to b" . The notation S_I is adopted as an abbreviation for S_R^{-1}, the inverse of S_R.

GSL	Reversible Substitution S_R	Inverse Substitution S_I
$v := e$	push v ; $v := e$	pop v
$S\,;\ T$	$S_R\,;\ T_R$	$T_I\,;\ S_I$
$g \Longrightarrow S$; $\neg g \Longrightarrow T$	push b; $g \Longrightarrow$ (S_R ; push true); $\neg g \Longrightarrow$ (T_R ; push false)	pop b; $b \Longrightarrow S_I$; $\neg b \Longrightarrow T_I$; pop b
$S\,[]\,T$	push b; (S_R ; push true)[] (T_R ; push false)	pop b; $b \Longrightarrow S_I$; $\neg b \Longrightarrow T_I$; pop b

TABLE 3.1: Reversible Substitutions in the style of Zuliani.

We note that the non-deterministic choice $S[]T$ shows a different interpretation from ours; in the table, the reversible version merely records the choice taken and the inverse then reverses through that. This kind of reversing recalls Bennett's model in which the whole program reverses, using its own history. There is no notion of taking the other choice, as Zuliani's model lacks any semantic mechanism to trigger reversal, whereas we use stand-alone guards to allow infeasibility to do this. The choice substitution will then try another choice, thus we have a finer-grained reversibility to allow provisional choices.

Let us see if we can modify Zuliani's approach here to construct a stepwise reversible operation by using the form $x := x + e$, which has the inverse $x := x - e$. The exchange of two values in two locations is also reversible (in fact it is its own inverse); we write this as a parallel assignment

$$x := y \parallel y := x$$

The role of the stack can be taken by an integer array h with index i (where h and i are fresh variables with respect to the program), so $h(i)$ is the top of the stack. Initialising this stack and i is the initial cost referred to in the previous section.

Our reversible transformation of $x := e$ is stepped through in the table below; the zero in $h(i)$ at the start is its initialised value, and "?" indicates an unknown stack value. The initial value of x is shown as x_0.

Assignment	$h(i-1)$	$h(i)$	x
	?	0	x_0
$h(i) := h(i) + e$	?	e	x_0
$x := h(i) \parallel h(i) := x$	?	x_0	e
$i := i + 1$	x_0	0	e

The reverse operation is formed from the inverses of each of the above steps.

Assignment	$h(i-1)$	$h(i)$	x
	x_0	0	e
$i := i - 1$	?	x_0	e
$x := h(i) \parallel h(i) := x$	?	e	x_0
$h(i) := h(i) - e$	?	0	x_0

We therefore see that stepwise reversible operations can be constructed, though at the cost of some increased complexity. For this reason we have not strictly followed this path in the RVM, opting for "semantic reversibility" as a more efficient and achievable compromise. For this, it is sufficient that a program yield the same results when reversing as would a stepwise logically reversible program. Irreversible "internal" steps are tolerated if the overall effect of the operation is the same, and also we need not initialise unused stack locations, as they have no effect on the program.

3.3 Summary

In this chapter we have addressed Objective 1; we surveyed the background of Reversible Computing, showing how it provides a possible solution to the increasing problem of heat generation in the ultra-small, ultra-fast processors of the near future. We discussed how programming for reversible systems will require new programming paradigms, demonstrating an assignment statement with stepwise reversible instructions which could exploit reversible hardware and thus eliminate most irreversible steps. We described our concept of semantic reversibility as being more appropriate for a higher-level approach to the issue of programming for reversible hardware.

We turn our attention in the next chapter to the question of how we integrate these ideas on reversibility into the formal specification language at its implementation level.

Chapter 4

Reversible B

4.1 B0, RB0, and RB1

There are two basic versions of the language used in classical B, both based in GSL but one having a syntactic sugar coating, the Abstract Machine Notation AMN. For the abstract machines there is a version which allows non-determinism, sets and set operations, and functions defined as abstract constants. These properties, along with the fact that the specification language is not a procedural language — there is no mechanism for sequential composition — prevent its being suitable for translation to most executable computer languages.

When the specifications have been refined to implementations, this language will have been replaced by B0, which *can* be translated to languages like C and Ada. B0 is procedural in nature, and does not have non-deterministic elements or set facilities. Any functional constants will be replaced by operations, using appropriate structures; for instance, recursion would be replaced by loops.

The restriction that B0 only use constructs that can be translated to code becomes far less restrictive when the code is RVM Forth. We examine the RVM language in our next chapter, but here we point out that it implements sets and set operations directly, while in addition supporting various forms of non-determinism with regard to assignments and choice, and lambda calculus. So these methods no longer have to be refined away before translation to code. The RVM also supports certain constructs arising from reversibility, which will be described in this chapter.

The proposed language RB0 (Reversible B0) is an implementation-level language, all constructs of which can be translated to RVM. It may not necessarily be the most efficient RVM, as RB0 is primarily geared towards program proofs, and hence is mostly mathematical in structure. It is based on pure GSL, although certain AMN constructs are retained herein for comprehensibility (such as variable declarations with **VAR**). The AMN component visible in RB0 listings in the current thesis may not be required in the future, as RB1 — described below — is the natural home for such features. Some non-deterministic strategies disallowed in B0 can be accommodated in RB0, though we cannot accommodate unbounded non-determinism.

There are also some additions, providing an enriched formal semantics; in particular the guard $\Longrightarrow$, which can now occur in a stand-alone form (described in the next section), thus allowing infeasible situations as part of the reversibility semantics. Also we have the construction $S \diamond E$, described in section 4.4. It is hoped that increasing Unicode capability in editors and proof environments (and of course fonts) will eventually allow the use of such symbols directly, rather than as ASCII equivalents.

A further addition is to allow functions to be concrete constants, as described in chapter 6, section 6.1. Some notation from bunch theory can be used here to provide a translatable functional language for RB0.

In order that the translations to code be improved, the proposed language RB1 is under development. It will be an extension of RB0 which can be easily converted back to RB0. But it will contain additional information, not required for proof, to facilitate the translation to code, preferably more optimised and efficient code. The extra information might include details such as whether or not variables are to be declared reversible, and more accessible type information; as is seen in the chapter on translation schemas, this is not always easy to get across. An example of a proposed construct in RB1 is preferential choice (currently under investigation and development). In GSL and RB0, a choice such as $S \, [] \, T$ is symmetrical, with both S and T having an equal chance of being chosen first. But situations will arise where one would want S to be tried first, with T as a backup in case S proved infeasible. The way the RVM makes choices in such a scenario is tailor-made for implementing preferential choice; a provisional notation is

$$S \rhd T$$

which favours S as the first choice. As far as proof obligations go, these are exactly the same as those for symmetrical choice, however there is an extra semantic layer wherein the choice is directed.

The full description of RB0 and RB1 is beyond the scope of this thesis. Until RB1 is closer to completion, we must regard the fine details of RB0 as remaining somewhat fluid, particularly the extent to which AMN is retained. Ideally, in translating RB1 to RB0, the result should be closer to pure GSL, with only required extensions.

4.2 Stand-alone guarded commands and choice

We discussed the role of the guard in providing conditional constructs in section 2.1.5, and what might happen if a false condition arose with no forward path through an ELSE — whether implicit or explicit.

One possibility is to treat this as an infeasible situation; the program, being unable to go forward, goes into reverse to try another path. There must therefore be other paths available to try, and these are provided by *choice* constructs.

The basic use for these is non-deterministic — when specifying we have no way of knowing which choice will be made — and the state of the program at the point of choice is the state to which it will return on encountering an infeasible guard.

Various kinds of choice can be made, from simply assigning a variable from a set of values, to a broader choice such as which search strategy to embark upon. If these choices are returned to as the result of a reversal, as they were found to lead to infeasible situations, the program can assign an unused value to the variable, or attempt another search strategy, for instance.

At some point, if no feasible route is found, the available choices in any situation will run out, and now the program must continue in reverse mode until the previous choice point is reached, assuming there was one. In each case, the state *before* the original choice was made must be restored. Trying another path forward from here now makes all future choices available again at their appropriate points in the program.

Should *all* the choices run out of options, that would mean the whole program is infeasible. What happens here is dependent on the implementation, so an error message might be displayed, or an exception raised, or execution may simply cease with a particular return code.

4.3 Backtracking interpretation

An operational interpretation of the above is that the program is backtracking to try another choice; this might otherwise be accomplished with a recursive algorithm, for instance.

A simple example (this might represent a fragment of a larger program) featuring a choice between two assignments is

$$\begin{aligned}&x := 1 \,[]\, x := 2;\\ &x = 2 \Longrightarrow \mathsf{skip}\end{aligned} \tag{4.1}$$

In this case, if the assignment to 1 is made, the guard condition is false, so the program has no forward path; it must go back to the choice to try the other assignment. A similar example in a slightly larger context is

$$\begin{aligned}&x := 10;\ x := x + 1 \,[]\, x := x + 2;\\ &x = 12 \Longrightarrow \mathsf{skip}\end{aligned}$$

In this case the original value of x must be restored (the method will depend on the implementation), so we note that the value has not been erased. A history of the values x has taken would in principle be maintained for the program's "past", for stepwise reversibility. Since we use semantic reversibility, this is not needed in such detail.

4.3.1 Example: Send More Money

We provide a longer example based on a word problem, which is to assign the values 0-9 to letters, with each letter a different value, in such a way that the sum below is arithmetically valid:

```
    SEND
+  MORE
= MONEY
```

The approach we choose here is to assign any two values to d and e, so these are the first points of choice. Now y will be assigned $(d + e) \pmod{10}$, and the first condition to check is that y is neither equal to d nor to e. If it is, then the program must go back and make another choice for e. Had d been assigned 0, then the condition would remain false through all the choices for e, and in the end the program would have to return to its first choice and make another choice for d. In the absence of a specific heuristic or shortcut (such as disallowing 0 for d), the mechanism takes the strain by trying every available value until something works.

Once the condition is satisfied, however, the program can then go on to assign two more values to n and r, and the conditions from then on become more and more dependent on previous choices. For instance the value of e chosen in the second choice must be compatible with the e in "send" for its position in the sum.

The initial specification in B simply gives the overall properties of the solution; a refinement might break this up into conditions for each letter, and the implementation provides a procedure for achieving the solution, using the stand-alone guard to retreat from infeasible paths. However, the current B implementation language cannot be used here, as it does not allow non-determinism (specifically, assigning any number from a set), nor does it have notation for a stand-alone guard; so we use RB0.

In the RB0 implementation in figure 4.1, we use the GSL symbol for guard "$\Longrightarrow$", and ":$\in$"for non-deterministic assignment from a set, which are the points where a choice is made. The set $DIGIT = \{0, 1, \ldots, 9\}$, and the variables c_0, c_1, c_2 are the carry values from the additions.

4.4 Introducing a formalism to provide all results of a search ($S \diamond E$)

So we have a way of finding a viable path through a program, and avoiding infeasible ones. However, there may well be more than one path — that is, more than one solution to a given problem, more than one result for a search, and so on. An extension of the ideas described above is to have the program explore *all* the paths it can take, and storing each result in an output set. This entails reversing after each successful completion, and restoring the state. If the program is part of a larger program, we note that as this is also done after the final run, the inner program can be run and its possible outputs collected without affecting the state of the outer program.

Rather than using sets to reason about these issues, our theoretical development uses bunches, as described in section 2.2, to simplify the exploration of more complex applications. The approach is described in terms of prospective values, as it produces all the values which would have eventually resulted if the program

```
**solution** $\hat{=}$
  **BEGIN**
  **VAR** $c_0, c_1, c_2$ **IN**
    $d :\in DIGIT; \quad DIGIT := DIGIT - \{d\}$
    $e :\in DIGIT; \quad DIGIT := DIGIT - \{e\}$
    $y := (d + e) \bmod 10;$
    $c_0 := (d + e)/10;$
    $y \in DIGIT \Longrightarrow$
      $(DIGIT := DIGIT - \{y\};$
      $n :\in DIGIT; \quad DIGIT := DIGIT - \{n\};$
      $r :\in DIGIT; \quad DIGIT := DIGIT - \{r\};$
      $(n + r + c_0) \bmod 10 = e \Longrightarrow$
        $(c_1 := (n + r + c_0)/10;$
        $o :\in DIGIT;$
        $(e + o + c_1) \bmod 10 = n \Longrightarrow$
          $(c_2 := (e + o + c_1)/10;$
          $DIGIT := DIGIT - \{o\};$
          $s :\in DIGIT;$
          $(s + M + c_2) \bmod 10 = 0 \Longrightarrow$
            $(M = (s + M + c_2)/10 \Longrightarrow \text{skip}))))$
  **END**
**END**
```

FIGURE 4.1: RB0 for Send More Money.

had been run non-deterministically on all possible paths. A Prospective Value Semantics (PV) has been developed in [33] to formally describe the approach, using the notation

$$S \diamond E$$

where S is a GSL substitution (program) and E is an expression. The construct is then the bunch of results obtained from running S, then evaluating E for all possible outputs. We refer to PV semantics, PV computations, and PV facilities or constructs, and sometimes simply "the $S \diamond E$ construct". The implementation platform (RVM) has no internal representation of bunches; such bunches of results are packaged in a set for further use by the program. For consistency and translatability, the implementation-level specification in RB0 also packages bunches as sets in this way (by the simple expedient of putting set brackets around the diamond operation).

A simple example is this (the choice has a higher precedence than the diamond):

$$x := 1 \; [] \; x := 2 \diamond 5x$$

Note that this is only a descriptive mathematical syntax, rather than RB0 (hence the lack of set brackets). Were this to be just run sequentially, with the $5x$ following the choice, we would have one of two outputs, 5 or 10. The diamond construct forces the program to run again, taking whichever choice it did not take before. We thus have the two possible outputs $5, 10$ at the end.

Suppose we have an array a (size s) of natural numbers, and wish to find the position i in which a particular number e is stored (if it is); the latter is an input parameter. The following program would assign any position to the global variable i, test to see if the array contained e at that point, and if not keep trying (reversing if the guard was false).

$$\begin{aligned} &P(e) \;\hat{=} \\ &\quad i :\in 1..s; \\ &\quad a[i] = e \Longrightarrow \mathsf{skip} \end{aligned} \tag{4.2}$$

This would then leave one position at which e is found in the variable i, or if no such position exists, would reverse. In order to find all the positions which are occupied by the value e, we can use the diamond to collect all the results of the search:

$$P(e) \diamond i$$

This yields a bunch of all the applicable positions, or simply the null bunch if the element e was not in the array. In implementation, it would return a set which might be the empty set, and would be written in RB0 as $\{P(e) \diamond i\}$.

As a final brief example, suppose we have a set of sets of some particular type of element, and wish to flatten it to a single non-nested set. A B specification-level operation for this would describe the output in terms of a set comprehension:

```
$o \longleftarrow$ **flatten**($S$) $\hat{=}$
    **PRE** $S \in \mathbb{P}(\mathbb{P}(\mathbb{N}))$
    **THEN** $o := \{x \mid x \in \mathbb{N} \wedge \exists e.(e \in S \wedge x \in e)\}$
    **END**
```

Then an implementation-level refinement in RB0 using non-deterministic assignment and the diamond (which also has lower precedence than the sequential operator) would be

```
$o \longleftarrow$ **flatten**($S$) $\hat{=}$
    **VAR** $x, s$ **IN**
        $o := \{e :\in S;\ x :\in e \diamond x\}$                    (4.3)
    **END**
```

We examine the conversion of this to RVM code in a later section, as it is an example of a situation where RB1 could be used to specify a more efficient translation.

4.4.1 Referential Transparency

There are certain implications for the generation of proof obligations using PV semantics — although these are part of a wider semantic agenda than that addressed by this thesis. We should draw attention, however, to the particular case of Referential Transparency. This assumes a greater importance in the extended language for concrete functions discussed in chapter 6.

Referential transparency refers to the property that, when two expressions are equal, one can be replaced by another in an expression without affecting the result. More formally, where $\boldsymbol{E}, \boldsymbol{F}$ and $\boldsymbol{G}$ are expressions, and $G\langle E/x\rangle$ means replace x with $\boldsymbol{E}$ in $\boldsymbol{G}$, we have the inference rule

$$\frac{\mathsf{HYP} \vdash\ E = F,\ \mathsf{HYP} \vdash\ G\langle E/x\rangle}{\mathsf{HYP} \vdash\ G\langle F/x\rangle}$$

But this is not always the case in $\boldsymbol{S}\diamond\boldsymbol{E}$ constructions. A simple counter-example would be: given that $x = y \wedge x = 0$, compare

$$\begin{aligned} x := 1 \diamond x &= 1 \\ x := 1 \diamond y &= 0 \end{aligned}$$

The output in the second case is not the same as in the first.

In certain situations there will be no problem; for instance if the equality describes a mathematical identity between two expressions using the same variable. Thus syntactic substitutions based on, say,

$$\begin{aligned} &2y = y + y \qquad \text{or} \\ &x^2 = xx \end{aligned}$$

will behave blamelessly. We can write these in the stronger form of equivalences, $2y \equiv y + y$ and $x^2 \equiv xx$.

More generally, the substitution scope in a PV computation would need to be limited to one side of the diamond — either the output bunch, requiring equivalence, or the expression generating it. We can write these as in the following examples, where S is a GSL/RB0 substitution and H is another expression.

$$\frac{\mathsf{HYP} \vdash \; E \equiv F}{\mathsf{HYP} \vdash \; S \diamond G\langle E/x\rangle \quad = \quad S \diamond G\langle F/x\rangle}$$

$$\frac{\mathsf{HYP} \vdash \; E = F}{\mathsf{HYP} \vdash \; x := G\langle E/x\rangle \diamond H \quad = \quad x := G\langle F/x\rangle \diamond H}$$

4.5 Case Study: Sudoku solver

As a more sustained demonstration of the techniques of modelling with sets and sequences and specifying using our reversibility facilities for automatic backtracking, we investigate a solver for Sudoku, the number puzzle. **Note:** Some of the material in this section and section 5.4 is based on an early treatment in a paper given at the 21st Euroforth conference [16].

The basic form of this puzzle is a 9×9 grid of squares divided into nine 3×3 subsections, hereinafter called "sectors". A puzzle consists of a partly filled-in grid which must be filled in such that every row, column and sector contains all the digits 1 to 9.

The structure of the puzzle and its solutions leads naturally to a set-based model, wherein the constraints on a square's value and the properties of a solution can be readily expressed. For instance, each blank square has an associated set of numbers which are available to it; as the grid fills up, this set will shrink — but if a blank square should find itself with *no* available values left, this means that the grid cannot be completed.

As the notation will involve a certain amount of RB0, particularly in the implementations (and simplifications for readability), this is neither a syntactically viable nor complete development in B. In particular the interface details are left to the code stage, and the RVM code will be examined in section 5.4.

4.5.1 Data Model

The basic requirement to represent the grid is a mapping from each square which has so far been assigned to its assigned value; and for the unassigned squares, a mapping to a set of their available values. The values 1 to 9 are defined as the set *DIGIT*. A square is defined by row-column co-ordinates; the set *XY* is 0 to 8, and the definition *SQUARE* is a maximal set used as a type, consisting of ordered pairs of the numbers 0 to 8 denoting *(row,column)*.

So the grid is a partial function:

$$grid \in SQUARE \nrightarrow DIGIT$$

Complementary to this is a function from the blank squares to the set of their possible values, this being of type:

$$possible \in SQUARE \pfun \mathbb{P}(DIGIT)$$

The elements in these respective sets thus have the following forms (for row r, column c, and value v_i, $1 \leq i \leq n$ for n available values).

An element of *grid* has the form $(r \mapsto c) \mapsto v$
An element of *possible* has the form $(r \mapsto c) \mapsto \{v_1, v_2, \ldots, v_n\}$

Since B reserves the notation (r, c) for variable or parameter/ argument lists, we use the maplet symbol $\mapsto$ to denote ordered pairs. This also avoids any confusion with bunch notation, wherein the comma is a union symbol.

As the game progresses, the domain of *grid* grows while that of *possible* shrinks. On successful completion, *grid* will become a total function. However, should the set associated with a particular blank square in *possible* become empty, this means that the grid cannot be filled, and completion of the game is now infeasible. This is used as the guard for backtracking with the reversibility mechanism.

The elements so far discussed are incorporated in an initial machine called *grids*, which is parameterised — it takes a starting grid called *puzzle*, and the **CONSTRAINTS** clause ensures that this is of the correct type. There are no operations in this machine.

```
MACHINE    $grids(puzzle)$
SETS
  $DIGIT = \{1,2,3,4,5,6,7,8,9\}$
  $XY = \{0,1,2,3,4,5,6,7,8\}$
DEFINITIONS
  $SQUARE \;\hat{=}\; XY \times XY$
CONSTRAINTS
  $puzzle \in SQUARE \pfun DIGIT$
VARIABLES    $grid,\ possible$
INVARIANT
  $grid \in SQUARE \pfun DIGIT \wedge$
  $possible \in SQUARE \pfun \mathbb{P}(DIGIT)$
INITIALISATION
  $grid := puzzle \parallel possible := \varnothing$
END
```

We now need to consider the constraints on what values can be entered in blank squares, and the generation and maintenance of the set *possible*.

4.5.2 Constraint Zone

The set of possible values for a square are those values which are not currently used in its row, column, or sector; we define these three together as the Constraint Zone for the square, illustrated in figure 4.2.

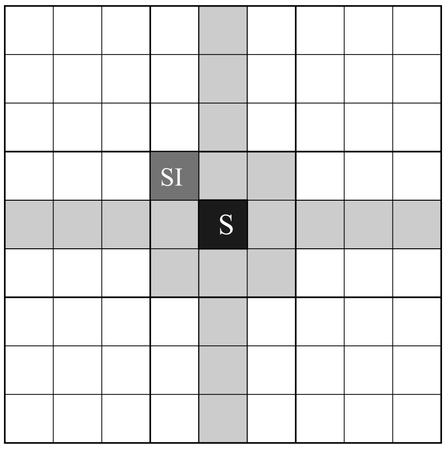

FIGURE 4.2: The constraint zone for square *S*.

To construct the zone for any square, we construct the row, column and sector separately, and define the zone as the union of these. The constructions use functions declared as constants, whose properties define their action. We examine more closely the issues of constant functions later in section 6.1.

The row and column are simply generated from the appropriate member of the ordered pair for a square; for instance in the specification (a complete listing for the machine follows the descriptions below), a constant function *genrow* is defined with these properties:

$$\begin{aligned}&genrow \in XY \rightarrow \mathbb{P}(SQUARE) \land \\ &\forall\, r.(r \in XY \Rightarrow genrow(r) = \{r\} \times XY)\end{aligned}$$

The specification states the type of the function, and it generates from the row number a set of all the squares in that row. The column generator is almost identical, but the output set has the column number as the second element in each pair.

The square labelled SI in figure 4.2 is a sector index (the top left-hand square of each sector), which aids the construction of a sector set for any square. The constant *corner* maps any row or col number to that of the sector index, and from this row and column, the whole sector can be generated by the function *sector*. The union of this with the row and column is a simple definition in the specification, although an RVM word will be defined to cover this.

Having generated our Constraint Zone, the relational image of these squares with the relation *grid* will yield the subset of *DIGIT* that the square **cannot** be assigned, and the complement of this set is paired with the square as the output of the constant function *available*.

With much of the work being done by functions, only one operation is required at this stage, that which builds the initial version of *possible*. The abstract specification does this with a set comprehension, but the implementation refinement makes use of the diamond operator as described in section 4.4.

The machine, then, is called *czones* and listed below. The notation $m..n$, where $m < n$, refers to the set of natural numbers $\{m, m+1, \ldots, n\}$.

MACHINE *czones*

INCLUDES *grids*

CONSTANTS

$\quad row,\ col,\ genrow,\ gencol,\ corner,\ sector,\ available$

DEFINITIONS

$\quad czone(r,c)\ \hat{=}\ genrow(r) \cup gencol(c) \cup sector(r,c)$

PROPERTIES

$row \in SQUARE \rightarrow XY\ \wedge$

$\quad \forall (r,c).(r \in XY \wedge c \in XY \Rightarrow row(r \mapsto c) = r\ \wedge$

$col \in SQUARE \rightarrow XY\ \wedge$

$\quad \forall (r,c).(r \in XY \wedge c \in XY \Rightarrow col(r \mapsto c) = c\ \wedge$

$genrow \in XY \rightarrow \mathbb{P}(SQUARE)\ \wedge$

$\quad \forall\, r.(r \in XY \Rightarrow row(r) = \{r\} \times XY)\ \wedge$

$gencol \in XY \rightarrow \mathbb{P}(SQUARE)\ \wedge$

$\quad \forall\, c.(c \in XY \Rightarrow col(c) = XY \times \{c\})\ \wedge$

$corner \in XY \rightarrow XY\ \wedge\ \forall\, n.(n \in XY \Rightarrow corner(n) = n - (n\ \mathsf{mod}\ 3)\ \wedge$

$sector \in SQUARE \mathrel{+\!\!\!\rightarrow} \mathbb{P}(SQUARE)\ \wedge\ \forall (r,c).(r \in XY \wedge c \in XY \Rightarrow$

$\quad sector(r,c) = (corner(r)..corner(r)+2) \times (corner(c)..corner(c)+2))$

$available \in SQUARE \mathrel{+\!\!\!\rightarrow} \mathbb{P}(DIGIT)\ \wedge\ \forall (r,c).(r \in XY \wedge c \in XY \Rightarrow$

$\quad available(r,c) = DIGIT - grid[czone(r,c)])$

OPERATIONS

init_possible $\hat{=}$

$\quad possible := \{(r \mapsto c) \mapsto V \mid$

$r \in XY \wedge c \in XY \wedge V \in \mathbb{P}(DIGIT)\ \wedge$

$\qquad (r \mapsto c) \in SQUARE - \mathsf{dom}(grid) \wedge V = available(r,c)\}$

END

The abstract version of the operation **init_possible** is merely a static description of the completed set, but the implementation version is as follows. As this works directly with a square as a maplet, it requires the "splitter" functions *row* and *col* to isolate the row and column for input to *available*.

```
init_possible $\mathrel{\widehat{=}}$
  VAR $S, s, V$ IN
    $S := SQUARE - \mathsf{dom}(grid)$;
    $possible := \{s :\in S;\ V := available(row(s), col(s)) \diamond s \mapsto V\}$
  END
```

Having obtained the set of blank squares S, this operation takes a choice of a square s from S, applies the function *available* to it, and constructs a pair from the square and its available set of values.

The diamond operator then forces backtracking to choose another square, which is then paired with its own set of values, and so on until there are no more unchosen squares. This results in a bunch of maplets, and as the operations take place inside set brackets, a set is the final result.

Naturally the same output could be achieved with a loop, however the formal semantics of a loop require a number of proof obligations relating to loop invariants and termination. The PV version has no effect on state except for the assignment at the end, the semantics ensure termination for a finite set, and all that should be required is that the set produced is of the same type as *possible*.

4.5.3 Method

The above gives us the main elements of the static model, but we need to specify the operations which will assign values to squares, update the grid and the function *possible*, and deal with the situation where completion becomes infeasible.

We use one optimisation in the method, namely, we choose the next square from those most constrained (i.e. those with fewest available values left to assign). Constructing this provides a good illustration of the $S \diamond E$ facility.

The algorithm for "playing" the game is as follows:

1. Extract set of most constrained blank squares, along with their values (this is a subset of *possible*).
2. Choose (*non-reversibly*) a square from the domain of this set. Remove this entry from *possible*.
3. Choose a value from those available; this must be a reversible choice. Create a pair from the square and value, and add it to *grid*.
4. Update the blanks in the constraint zone of the square, i.e. remove the value just assigned from their available sets.
5. If any of the resulting sets are empty, the grid is now non-viable; in this case, we must backtrack to step 3, restoring state along the way.
6. Otherwise, update *possible* itself with these new values.
7. Repeat until grid full.

4.5.4 Algorithm expansion

1. Concentrating on the most constrained blanks simply seems most logical; should a wrong choice be made, there will be fewer subsequent attempts to work through. More sophisticated techniques would be able to reduce the available values by other comparisons with the current state; our naïve version does not apply all possible constraints, but instead allows the backtracking mechanism to take the strain.

2. The above being the case, it is likely that more than one square will be returned, so one must be chosen to work with. A point which bears stressing is that, if a solution exists at all from this stage, it can be found from *any* of these squares. Faster from some than others, perhaps, though there is no way of knowing which. So:

   (a) The choice may as well be non-deterministic.

   (b) The choice *must not* be reversible.

   The latter point may well not be obvious at first glance (it can be tempting to assume that any choice should be reversible). However, if none of the values from the chosen square lead to a viable grid, then reversing will simply cause another square from the set (if any) to be chosen. But this will also fail, needlessly embarking on a fruitless and perhaps lengthy search, as the ultimate non-existence of a solution from that configuration may take some time to become apparent. If this is repeated every time there is a choice of squares, the search space is expanded to an enormous degree. This choice is an example of inappropriate use of a reversible choice, since there is no division into “wrong” and “right” squares, and therefore nothing to be gained from trying another one on failure.

3. The choice of value, on the other hand, is clearly critical: probably only one will lead to a solution.; this should be the only reversible choice in the main program. If no valid number exists, then a previous assignment must have been wrong.

4. The assigned value is now no longer available to those blanks constrained by the current square, so it must be removed from all of their “possible” assignment sets.

5. This should always leave at least one remaining possible value for a square; an empty set here indicates we cannot find a solution given this assignment. This is the test for the reversibility guard, which will provoke backtracking.

6. The action guarded is updating the *possible* set (described in more detail below).

4.5.5 Most Constrained Squares

As described above, we have a function from each blank square to the set of values available for it, called *possible*.

We would like to choose our next square from those with the smallest of these sets. Our way of doing this is to build a set of such squares, and choose one (non-reversibly as described). For the specification, one preliminary element is a constant function *mcn*, which returns the size of the smallest set of available numbers.

$$
\begin{aligned}
&mcn \in (SQUARE \nrightarrow \mathbb{P}(DIGIT)) \to \mathbb{N} \wedge \\
&\forall poss.(poss \in SQUARE \nrightarrow \mathbb{P}(DIGIT) \Rightarrow \\
&\quad mcn(poss) = \mathsf{min}(\{\mathsf{card}(s) \mid s \in \mathsf{ran}(poss)\}))
\end{aligned}
$$

Owing to the implementation structure of sets in the RVM, a shortcut can be utilised for this, though a formal description is required at this stage.

It would be useful if B had a general facility for extracting the first and second parts of a pair; but as this is not the case we need to define constants to do this for both an element of *grid* and one of *possible*, as they are of different types. This makes four functions in all (in RVM we can collapse this to a more general pair of FIRST and SECOND operations). For example, to extract the second part of an element of *possible*, thus returning a set of type *DIGIT*, we define the constant *poss2* with properties as follows.

$$
\begin{aligned}
&poss2 \in SQUARE \times \mathbb{P}(DIGIT) \to \mathbb{P}(DIGIT) \wedge \\
&\forall (s, V).(s \in SQUARE \wedge V \in \mathbb{P}(DIGIT) \wedge s \mapsto V \in SQUARE \times \mathbb{P}(DIGIT) \Rightarrow \\
&\quad poss2(s \mapsto V) = V)
\end{aligned}
$$

And similarly for the other three, which are *grid1*, *grid2*, and *poss1*.

Along with *row* and *col*, this makes three pairs of functions we have had to define to perform the same actions on a pair. What would be a good facility for RB0 would be a polymorphic version of pair-splitting functions which worked on any type, the way set union or maplet construction work on sets or elements of any type. In section 5.4, we will assume the existence of such things, to make the correspondence with RVM code clearer. In this section we use the differentiated definitions to remind ourselves which kinds of objects we are dealing with.

The operation itself, *getsquares*, in the abstract specification uses ordinary set comprehension to describe the required set of squares.

$$
\begin{aligned}
&mcset \longleftarrow \textbf{getsquares} \;\hat{=} \\
&\quad mcset := \{x \mid x \in possible \wedge \mathsf{card}(poss2(x)) = mcn(possible)\}
\end{aligned}
$$

An implementation-level refinement which more closely resembles the final code is in RB0, using non-deterministic assignment from a set, a stand-alone guard and the diamond construct:

```
$mcset \longleftarrow$ **getsquares** $\hat{=}$
**VAR** $x, n$ **IN**
  $n := mcn(possible)$;
  $mcset := \{x :\in possible;$
    $\mathsf{card}(poss2(x)) = n \Longrightarrow \mathsf{skip} \diamond\ x\}$
**END**
```

This uses a guard to "filter" out those squares whose associated sets have the minimum number of members, the relevant part being

$$\mathsf{card}(poss2(x)) = n \Longrightarrow \mathsf{skip} \diamond\ x$$

Should the cardinality be other than n, the guard ensures backtracking to make another choice of x. So only those x for which the set meets the condition will be in the output set.

4.5.6 Machine *moves*

We look at our next machine, at the level just below top-level, which we call *moves*. It encapsulates most of the elements of the method, including those discussed above.

After we construct our set of most constrained squares, the next stage is to make a non-reversible choice from it. This being the case, it might be noted that there is in fact no need to construct this set — the first element x to meet the condition in the above operation would do as well as any other. But since our purpose is more illustrative than to provide a fully-optimised algorithm, we let it stand.

The usual translation of non-deterministic assignment from a set ($:\in$) is as a reversible choice, which is not wanted here for reasons explained earlier (page 38). So there is a certain notational problem in specifying this. For now we will use the substitution

$$x := \mathsf{choice}(S), \qquad \text{where for any set } S,\ \mathsf{choice}(S) \in S.$$

where $\mathsf{choice}(S)$ is a non-deterministic choice from the set S, but the corresponding translation into RVM is not reversible.

As we have seen the more important peripheral aspects of the specification machine, we go straight to the implementations of the operations, the abstract versions not being particularly edifying at this stage.

The operation **nextup** takes the output from **getsquares**, and makes a non-reversible choice of square + set of values, which is then the input for the operation **assign**; this makes a *reversible* choice of value from the set of values, constructs a maplet of square + value, and adds it to *grid*, the set of filled-in squares. This square + value is stored in the machine variable *assigned*.

The next task is to update those elements of *possible* which pertain to the blanks in the constraint zone of *assigned*, as its value is now no longer available to them.

```
OPERATIONS    ( extract from Implementation $movesI$ )
$mcset \longleftarrow$ getsquares $\hat{=}$
VAR $x, n$ IN
   $n := mcn(possible)$;
   $mcset := \{x :\in possible;$
       $\mathsf{card}(poss2(x)) = n \Longrightarrow \mathsf{skip} \diamond\ x\}$
END;

$nx \longleftarrow$ nextup ($mset$) $\hat{=}$
VAR $x$ IN
   $x := \mathsf{choice}(mset)$;
   $possible := possible - \{x\}$;
   $nx := x$
END;

assign ($nx$) $\hat{=}$
VAR $n$ IN
   $n :\in poss2(nx)$;
   $assigned := grid1(nx) \mapsto n$;
   $grid := grid \cup \{assigned\}$
END;

update_possible $\hat{=}$
VAR $x, U, upd$ IN
   $U := czone(row(grid1(assigned)), col(grid1(assigned))) \lhd possible$;
   IF $U \neq \varnothing$ THEN
      $upd := \{x :\in U \diamond poss1(x) \mapsto poss2(x) - \{grid2(assigned)\}\}$;
      $\varnothing \notin \mathsf{ran}(upd) \Longrightarrow possible := possible \Leftarrow upd$
   END
END
```

The method of **update_possible** here involves extracting the value from *assigned* with *grid*2(*assigned*); then use domain restriction on *possible* with the set of blanks in the constraint zone of *assigned* to produce the subset of *possible* pertaining to those blank squares.

It is possible that the square just assigned is the last blank in its constraint zone, and no updating will be required, so the next step is only carried out if the set obtained is not empty.

We now construct a new set from this, where all the sets of values have had the just-assigned value removed if it was present, and this is the updated set that needs to be written to *possible*. However, if one of these sets of values is now empty, one square has no possible assignments and completion of the grid is no longer feasible. So this condition is guarded. If it is the case that the empty set is in the range of the updates, the machine reverses to the point at which the choice of value was made, the line

$$n :\in poss2(nx)$$

in **assign**, to choose another value (if any remain).

If all is well and there are no empty sets, we use functional override to update the set *possible*.

The remaining task would be to call these operations in a top-level machine, to make a move and update the status (checking if the game was complete). The specified implementation would need to work via local variables, along these lines:

```
  $mcset \longleftarrow getsquares;$
  $nx \longleftarrow nextup(mcset);$
  $assign(nx);$
  $update\_possible$
```

The RVM translation could perhaps be optimised to use something more streamlined, analogous to

```
  $assign(nextup(getsquares));$
  $update\_possible$
```

This constitutes one step of the solution — a move, as we characterise it. There will thus be a loop which repeats moves to fill the grid up. If an infeasibility is encountered, the first reversal will choose another value for the square, but if there are none left, the reversal must continue back to the last loop, to choose another value for the square that was assigned in that. It is possible that a whole branch of paths will prove infeasible, and the grid must go back through several loops to an earlier stage to choose another value.

4.6 Summary

In this chapter, we have described the changes in B0 and its equivalent GSL, which will provide the functionality required in RB0. This has mostly been GSL-based, but with certain AMN-style constructs, e.g. **IF-THEN** conditionals, and **VAR** declarations to aid readability. These, along with loops, will ultimately form part of RB1. Currently RB0 is in a slightly hybrid state in this regard, and the constructions used in this thesis are summarised in Appendix B. PV constructions have also been described, the foregoing achieving parts of Objective 2.

The use of the introduced constructs has been demonstrated in a formal setting in a number of small examples and the development of a larger-scale one, achieving parts of Objective 3 (the remainder of these objectives is addressed in the next chapter).

Having explored the structure of reversible specifications, in the next chapter we examine the target execution language for them, describing the extensions that make Forth into RVM-Forth; and in particular translate the Sudoku machines in section 5.4.

Chapter 5

An Implementation Platform for our Ideas (The RVM)

The basic features of Forth have been extended in a number of ways to render it suitable for the implementation of a Reversible Virtual Machine (RVM). Some of these extensions are to the interface — new facilities and constructs for use in programming and some are internal changes which allow for simulated reverse computation.

There are also certain extensions to allow implementation from a more abstract specification level, in particular a facility for defining sets and sequences, and performing operations on them. These operations include some non-deterministic ones, choice from a set for instance, which may or may not be reversible, and assignment from a set (this will generally be reversible).

A facility for local variables is provided, which is not part of Standard Forth, although some custom systems provide such a facility. However in this case the reversibility must be taken into account.

5.1 Differences in Interface and Programming Constructs

Following the discussion of guards and choice in the previous chapter, one would expect these to be provided. The code for example (4.3), page 28, with a choice between two assignments, and a guard, is as follows (with a small addition):

```
: T1  <CHOICE 1 to X
          [] 2 to X CHOICE> X . X 2 = --> ;
```

The addition is `X .` which simply prints out the current value of X. The output is either 2, or 1 followed by 2, as the assignment of 1 fails the condition in the guard.

```
T1 1 2 ok
```

In this case the implementation runs the choices in order from first to last; in the sense that this order is arbitrary when writing the code, the construct is non-deterministic. However, a random running order can be obtained with a wrapper using the `CASE` statement and the `RANDOM-CHOICE` word; this is shown in the schemas in section 7.6.2.

The general form for the choice construct, then, is

```
<CHOICE S1 [] S2 [] ... [] SN CHOICE>
```

where there are N choices of program S_i, $1 \leq i \leq N$.

The guard is written as `-->` . In the example above, the action after the guard is merely an unwritten `skip`, but normally code that required the guard to be true would follow it.

Suppose we alter the code for program `T1` to

```
: T2  <CHOICE 1 to X
          [] 2 to X CHOICE> X . X 3 = --> ;
```

This means that the guard cannot be true whatever runs prior to its invocation, so the output is

```
T2  1 2 ko
```

The `ko` at the end is a signal that no feasible path was found after every alternative had been tried.

The example “Send More Money” (RB0 code in figure 4.1) translates into a RVM program of a very similar structure, included as Appendix E.

In the translations from RB0 examined in this chapter, we normally use the RVM upper-case convention for operation names and variables; so a function *sector* becomes RVM `SECTOR`, an operation **getsquares** becomes `GETSQUARES`, and a variable *possible* becomes `POSSIBLE`. Also, underscores will become hyphens. There are a few exceptions to this, but we will quote the relevant RB0 alongside the translation to avoid any confusion.

Example (4.2), page 31, to search for an index matching a value in an array, would be specified with this RB0 (renaming the operation more descriptively), where s is the size of the array:

```
where_in_array(e)  ≙
  i :∈ 1..s;
  a[i] = e ⟹ skip
```

This converts to the following code, with some global set-ups for an array — with $s = 10$ — and an index variable. The target value is assigned to the local variable `N` (corresponding to e in the RB0), and the program must find this value on the stack. The value of the index found is left on the stack with `INDEX 1LEAVE`.

```
10 VALUE-ARRAY A
0 VALUE INDEX
HERE 10 , 45 , 25 , 78 , 25 , 78 , 33 , 78 , 25 , 69 , 44 , to A
```

```
: WHERE-IN-ARRAY ( n -- n )
    (: VALUE N :)
    1 10 .. RANDOM-CHOICE to INDEX
    INDEX of A N = -->
    INDEX 1LEAVE ;
```

If there is more than one instance of the goal value in the array, the value of the index at the first one found is returned, so we have two locations for 78. A search for a number not in the array will exit with the message `ko`.

A series of sample runs *should*, then, look like this:

```
69 WHERE-IN-ARRAY .  9 ok
33 WHERE-IN-ARRAY .  6 ok
78 WHERE-IN-ARRAY .  5 ok
78 WHERE-IN-ARRAY .  7 ok
88 WHERE-IN-ARRAY .  ko
```

However, actually running this can give the last two lines as

```
78 WHERE-IN-ARRAY .  7 ok
88 WHERE-IN-ARRAY .  3 ok
```

But we know 88 is *not* in the array, so what has happened? A feature of the internal organisation of the RVM is that a successful search for a value in this way will still leave choices to be taken on the history stack, along with the value of the relevant variable at the time. This is precisely the behaviour required in prospective value computations, which take place in a garbage-collecting wrapper, and for general reversibility in a larger context.

In this case, having found a location for 78, the index value is left on the stack, but the remaining choices are still kept. A search for 88 or any number not in the array now results in all the locations being searched; but when nothing is found, reverse execution *continues*, with the local variable having its last value restored, and another location of 78 being returned.

Running a stand-alone guarded command like this sequentially is less likely in a program, but can certainly arise interactively, as we see. The easiest way around this is to construct a wrapper for the search; the following example is an $S \diamond_1 E$ construct, which we describe in more detail on page 48 in section 5.1.1, after describing the $S \diamond E$ implementation itself.

```
: FIND-ELEMENT ( n -- n )
    (: VALUE N :)
    INT { <COLLECT N  WHERE-IN-ARRAY TILL CARD SATISFIED> } CHOICE ;
```

This leaves at most one value on the stack, and leaves no unresolved choices on the history stack. Thus we can interactively produce

```
78 FIND-ELEMENT .  3 ok
55 FIND-ELEMENT .  ko
```

5.1.1 Prospective Values Implementation

We can demonstrate the $S \diamond E$ construct using the `WHERE-IN-ARRAY` program, to find all the locations of a certain value in the array. The RB0 for this, based on example (4.2), is

$$\textbf{findall} \ \hat{=} \ \{where\_in\_array(e) \diamond i\}$$

The general form for the construct in RVM-Forth is

```
<set-type> { <RUN S E RUN> }
```

So the bunch of results is returned in a set. We examine the construction of sets and the available operations in more detail shortly, in section 5.1.2 — along with the stack signatures; those in the following few examples are not crucial. The construction can also be used without an enclosing set for deterministic computations where a single result is expected.

If S leaves its value for E on the stack, there is no need to do it explicitly as here. The code to find all the index locations, then, is

```
: FINDALL ( n -- n.P, a set of positions  )
    (: VALUE N :)
    INT { <RUN N WHERE-IN-ARRAY RUN> }
    1LEAVE ;
```

And some sample runs below — the word `.SET` prints a set.

```
25 FINDALL .SET  {2,4,8}ok
78 FINDALL .SET  {3,5,7}ok
88 FINDALL .SET  {}ok
```

An empty set is a perfectly valid return for the program in this case, so there is no `ko` if the value is not in the array.

An RVM program constructed literally from the specification in example (4.3), page 32 is as follows. The RB0 for this is

```
$o \longleftarrow \textbf{flatten}(S) \ \hat{=}$
    VAR $x, s$ IN
        $o := \{e :\in S;\ x :\in e \diamond x\}$
    END
```

The type of set S is obtained with `DEMO-ELEMENT`, and as `1LEAVE` leaves the constructed set on the stack, no assignment to an output variable like o is necessary. An automated translation schema would currently produce something along these lines.

```
: FLATTEN ( x.P.P -- x.P )
    (: VALUE S :)
    NULL VALUE E NULL VALUE X
    S DEMO-ELEMENT { <RUN
        S CHOICE to E E CHOICE to X X
    RUN> } 1LEAVE ;
```

It uses local variables x and s to match those in the specification; however, these are not really necessary as the stack can be used to store the values that `CHOICE` chooses. We can shorten the program by removing these (and the optional local `S`) in this way:

```
: FLATTEN ( x.P.P -- x.P )
DUP DEMO-ELEMENT { <RUN CHOICE CHOICE RUN> } NIP ;
```

The `NIP` removes an extraneous value from the stack, leaving only the set reference.

The output and the sort of set that it operates on are shown here on a subset of a power set `PSET`:

```
PSET .SET  {{3,4,5},{4,6,9},{5,6,7}}ok
PSET FLATTEN .SET  {3,4,5,6,7,9}ok
```

A variant of this form collects results until some criterion is satisfied, for instance a specific value is returned, or the set of results reaches a certain size. The form for this is

```
<COLLECT S TILL C SATISFIED>
```

in which `S` is an operation producing a result (generally one of several possible), and `C` is a condition on the set of results so far collected, returning a flag. When the flag is true, `SATISFIED>` stops the collection.

As this would be used inside a set constructor, the size of the set can be a condition — in particular a non-empty set will indicate that one result has been found. When one is all that is required, this provides a useful construct, formally $S \diamond_1 E$, which is implemented as

```
{ <COLLECT S TILL CARD SATISFIED> }
```

The type of the set would precede the first $\boxed{\{}$. This provided the wrapper in the `WHERE-IN-ARRAY` example on page 46.

The use of the provisional value construct *without* a set construction, for a single deterministic output, can be thought of as ascertaining what the result *would* be if the computation were to be run. Any state change, for instance assigning to a global variable, is undone when the reversal takes place. An example of this use is shown in section 6.1, page 70.

5.1.2 Sets and sequences in the RVM

The general declaration for creating a set in RVM is

<type> { <elem>$_1$, <elem>$_2$, <...> , <elem>$_n$ }

where the <elem>$_i$ are the values or declarations of elements. The comma $\boxed{,}$ is a word itself, which reserves storage in the set for each element. Thus the declaration

```
INT { 2 , 3 , 5 , 7 , 11 , 13 , }
```

would create the set $\{2, 3, 5, 7, 11, 13\}$ and leave a reference (to the heap) for the set on the stack. For contiguous sets of integers, there is a shortcut, which is the postfix version of the notation *m..n*; so `1 9 ..` creates the set of integers from 1 to 9.

The basic elemental types specified in <type> are `INT` and `STRING`. We have operations for set union, intersection and subtraction; these are respectively (for sets `S` and `T`)

```
S T \/   S T /\    S T \
```

The postfix as usual requires the arguments in the same order as the infix versions. An empty set of a specific type can be created with a declaration without any elements, and sets can be assigned to variables using `VALUE`, or constants, in the usual way .

To add or subtract a single element from a set, rather than create a new set with the element and use union or set subtraction, we provide the operations `ADD-ELEMENT` and `SUBTRACT-ELEMENT`, so obtaining

```
INT { 2 , 3 , 5 , 7 , 11 , 13 , }  17 ADD-ELEMENT  .SET (enter)
{2,3,5,7,11,13,17} ok
```

While order of elements is not important in set theory, the implementation sorts the elements numerically or alphabetically, or by size; this is mainly to facilitate searching, for instance checking membership with `IN` as in the following, which has assigned the name `P` to the set of primes above.

```
13 P IN . (enter)
-1 ok
```

More complex kinds of sets and elements are possible; for instance ordered pairs of elements can be created with the `|->` family of operations. In certain environments, this will deduce the types of the pair elements, however in others it may have to be augmented with type suffixes, for instance

```
" Bob"  4681 |->$,I
```

uses the operator with suffix `$,I` to create a (string, integer) pair. Pairs are also created when two sets have their cross product generated with `PROD`

```
STRING { " Bob" , " Alice" , } VALUE S  3 4 .. VALUE N
S N PROD .SET (enter)
{(Alice,3),(Alice,4),(Bob,3),(Bob,4)}ok
```

We can generate the power set of a set with `POW`, as in

```
S POW .SET (enter)  {{},{Alice},{Bob},{Alice,Bob}}ok
```

However, `POW` is more often used as a type specifier in building sets, along with `PROD`.

A set of maplets can be treated as a relation and perhaps a function. There are operations `DOM`, `RAN`, and `APPLY` to extract the domain and range, or to implement function application. Supposing we defined a function from names to employee numbers[1], then these operations yield

[1]Such numbers, like telephone numbers, would not normally be treated as integers.

```
STRING INT PROD { " Ponsonby" 668 |-> , " Blenkinsop" 792 |-> ,
" Proust" 333 |-> , } VALUE EMPS   (enter)  ok
EMPS .SET (enter) {(Blenkinsop,792),(Ponsonby,668),(Proust,333)}ok
EMPS " Ponsonby" APPLY .  (enter)  668 ok
EMPS RAN .SET  (enter)  {333,668,792}ok
EMPS DOM .SET  (enter)  {Blenkinsop,Ponsonby,Proust}ok
```

Note that the pair-building operator requires no extra type information in this situation, having "deduced" the types from the set specifier `STRING INT PROD`.

Such constructs can be nested to arbitrary depth. A function from a pair of integers to a set of integers, say, that is of type $\mathbb{N}\times\mathbb{N} \to \mathbb{P}(\mathbb{N})$ could be encapsulated in a set of type $\mathbb{P}((\mathbb{N}\times\mathbb{N})\times\mathbb{P}(\mathbb{N}))$. An element of such a set would have the form

$$((n,n),\{v_1,v_2,\ldots,v_n\})$$

The postfix version of such a type in a declaration is

```
INT INT PROD INT POW PROD {   ...
```

The final `{` here corresponds to the initial $\mathbb{P}$, since `{` obtains the type for the set it is about to build from an empty set on the stack. We also note no brackets are required in the postfix syntax as a set $\mathbb{P}(\mathbb{N}\times(\mathbb{N}\times\mathbb{P}(\mathbb{N})))$ would have had quite a different declaration.

Stack signatures for sets

The postfix form motivates the syntax for sets in stack signatures; the basic types were listed in table 2.4, page 17. Additionally the symbols $\boxed{\texttt{*}}$ and `P` are used to represent cross-product `PROD` and power set `POW`, while the components are separated by full stops for readability. So a set with elements of type `INT STRING PROD INT POW PROD` would be represented in a stack comment by

```
n.$.*.n.P.*.P
```

The final `P` defines the parameter as a set. The stack signature for `FLATTEN` in the previous section has

```
( x.P.P -- x.P )
```

which means that a set of sets of arbitrary type is the input, and a set of elements of that type is the output.

The parameters can be given labels to facilitate providing additional information, for instance `FLATTEN` could be described with

```
( s1:x.P.P -- s2:x.P, s2 is general union of all sets in s1 )
```

5.2 Internal Features

Detailed discussion of these would be too specialised for our purposes; we merely indicate briefly the main features to support the extensions described above.

The most important of these is the history stack, which accepts data required to restore earlier states of the system during reverse execution. Normal forward execution uses a return stack and a parameter stack (the normal user stack), and a frame pointer for local variables. The fundamental operations which deal with these structures are coded in Assembly language, while the higher-level operations are coded in Forth from these.

The basic operations include a number of words which manipulate memory and the system structures; most of these require a reversible version to support the reversible aspect. This version will typically be augmented to interact with the history stack, storing the actions undertaken in some way such that state can be subsequently restored.

Garbage collection is something of an issue where data structures such as sets are being manipulated; often temporary sets are created during the calculation of a result, perhaps the construction of another set or data structure. These temporary sets are then garbage, while the result is not, and would normally be preserved for reverse execution. Such data structures are accessed by reference, and the contents must be protected by copying them (e.g. by assignment to a variable).

The usual way to ensure garbage is reliably collected is to employ a three-stage process:

1. Run the calculation in forward mode.
2. Evaluate and copy the result or set of results.
3. Reverse the computation to collect all garbage generated by step 1.

We might compare this with Bennett's three-tape Turing machine described in section 3.1. The reverse computation here also frees history stack locations, and restores any variables to their original values.

The above is exactly the structure $S \diamond E$, so this can be used as a wrapper for complex computations to ensure that garbage is collected after the calculation has been run and any results saved.

5.3 Stack Use when translating to RVM from B (compared to C)

Here we return to the simple example from section 2.1.8, expanding the example to three small operations which are used in a calling machine. The development is included as Appendix D, but we will use simplified fragments in this section.

B operations can have a number of output parameters, whereas C functions only return a single value, and therefore must cope with B operations using passing and calling by reference. So in particular we look at how the use of a stack suggests a simpler parameter-passing mechanism for B operations.

We use a slightly altered version of the operation ***S***, along with simple operations ***T*** and ***U***, which are then used sequentially by the calling machine.

The original operation was

```
$b, c \longleftarrow$ S($a$) $\hat{=}$
    IF $a > 4$ THEN
        $b := 3a$;
        $c := 5a$
    ELSE
        $b := 3a$
    END
```

This can be translated into RVM using a similar model to the C translation in section 2.1.8, with pointers and referencing. In the following, the `n^` in the stack signature means that the address of an integer variable should be put on the stack; these would be declared elsewhere and, unlike in C, RVM variables are always initialised to some value.

```
: S ( n n^ n^ -- )
   (: VALUE a VALUE^ b VALUE^ c :)
   a 4 > IF
      a 3 * to b
      a 5 * to c
   ELSE
      a 3 * to b
   THEN ;
```

So given declarations

```
0 VALUE B   0 VALUE C
```

where `B` and `C` are initialised to 0, the above operation would be invoked thus:

```
7 addr B addr C S
```

The word `addr` puts the address of the following variable on the stack (one example of necessary prefix usage in RVM). After `S` runs, the two variables would have the values 21 and 35 respectively. However if 3 was substituted for 7 in the invocation, `B` would be 9 while `C` would remain as 0.

However, the RVM need not, as we mentioned earlier, be constrained to parameter passing by reference. Using the stack for input and output values (with local variables to facilitate manipulation inside the operation), the above operation could be translated in this way:

```
: S ( n -- n n )
   (: VALUE a  :)
   0 VALUE b 0 VALUE c
   a 4 > IF
      a 3 * to b
      a 5 * to c
   ELSE
      a 3 * to b
   THEN  b c ;
```

There are now no addresses involved, nor any external variables — the two output values are merely left on the stack. The variables b and c are locally declared and initialised. In many cases, they would not actually be necessary; certainly this particular example could be optimised to remove them. On the other hand, where more complex manoeuvres take place (as in the next example) they will be an aid in keeping track of what is occurring.

Also, when considering automated translations, a consistent way of handling operation declarations and their actions would be to always use locals, and then use other means (perhaps also automated) to optimise the resulting code.

5.3.1 Calling Imported Operations

In B, operations such as the one above can be called from a machine or implementation which imports them, and sequentially composed (a semicolon is the symbol for this). To illustrate this point, we first define two extra one-line operations,

$$y \longleftarrow \mathbf{T}(x) \;\hat{=}\; y := x^2$$
$$d \longleftarrow \mathbf{U}(e{,}f) \;\hat{=}\; d := e + f$$

Now we construct an implementation which imports the one containing S, T and U, containing an operation V, which calls these three operations. The **VAR** construct introduces and scopes local variables.

```
$r \longleftarrow \mathbf{V}(a) \;\hat{=}$
$\mathbf{VAR}\ p, q, s\ \mathbf{IN}$
$p, q \longleftarrow S(a);$
$s \longleftarrow T(q);$
$r \longleftarrow U(p, s)$
$\mathbf{END}$
```

The end result r of this is $3a + 25a^2$ but only if $a > 4$. As one might expect, the B-Toolkit is unable to discharge its proof obligations in this respect, which can be exemplified by

$$a \leq 4 \Rightarrow q \in \mathbb{N}$$

There are a further three proof obligations which cascade from this. So here, if the guard fails, the very type of the local variable q is called into doubt. The C code produced from this is

```
void V(int *r, int a) {
   int p, q, s;
   S(&p, &q, a);
   T(&s, q);
   U(r, p, s);
}
```

As the C uses pass-by-reference, there is now a layer of additional complexity in that addresses must be passed to the inner operations (and the theoretical passing of `&(*r)` in the last line becomes simply `r`). Naturally, as C does not initialise the locals, the output when the guard is false is undefined.

A translation into RVM following this pattern would be

```
: V ( n n^ -- )
   (: VALUE a VALUE^ r :)
   0 VALUE p 0 VALUE q 0 VALUE s
   a addr p addr q S
   q addr s T
   p s addr r U ;
```

However a more straightforward version which simply leaves the output on the stack is possible. The original operations are also translated in this way, so they do not require addresses to be passed, again leaving their outputs on the stack.

```
: V ( n  --  n )
   (: VALUE a  :)
   0 VALUE r
   0 VALUE p 0 VALUE q 0 VALUE s
   a S to q to p
   q T to s
   p s U to r r  ;
```

This would be possible to rework using the stack only, without locals; but it would be a considerable undertaking to design an automated system that could work out the necessary manipulations to interleave the outputs and inputs correctly.

5.4 Case Study: Sudoku solver revisited

In section 4.5 we developed a pseudo-B specification for a Sudoku program, with some implementation-level specifications in RB0. We now translate some of these into RVM to demonstrate how readily they can map onto RVM code without requiring extensive reworking of the data model, removal of non-determinism, or unnecessary loops. The reversible facilities are also demonstrated.

The initial state of the game is with a partly-filled grid; we required ways to generate the constraint zone for each blank square and the first configuration of the set *possible*, where each element is a pair comprising a square (itself a pair of co-ordinates) and a set of n possible values it can take, that is $((r \mapsto c) \mapsto V)$, where $V = \{v_1, v_2, \ldots, v_n\}$.

In the machine *czones* on page 36 we had a number of short functions defined as constants to generate the components and achieve the stages in building *possible*. Unpacking a square into its row and column is done in RVM with `FIRST` and `SECOND`, which work on pairs of all types, as they find the type information within the pair.

The next two functions are those which generate a complete row or column given its number (output is a set of maplets). The row generator for instance has this as its main line:

$$genrow(r) = \{r\} \times XY$$

where XY is the set 0 to 8 for co-ordinates (defined in RVM as `: XY 0 8 .. ;`).

This translates into an RVM operation `GENROW` with the code:

```
: GENROW ( n -- n.n.*.P  )
   (: VALUE r :)
   INT { r , } XY PROD ;
```

In this, the word `INT` is the type of the set, the comma in the set brackets stores the value r which is on the stack — creating the set $\{r\}$, and the word `PROD` generates the cartesian product of this with `XY`. The output from this operation looks like this

```
5 GENROW .SET
{(5,0),(5,1),(5,2),(5,3),(5,4),(5,5),(5,6),(5,7),(5,8)} ok
```

The `GENCOL` operation is very similar, but with the constant as the second element in the pair.

Then the sector is produced from the row and column, using the utility *corner*, in this way

$$sector(r, c) = (corner(r)..corner(r) + 2) \times (corner(c)..corner(c) + 2))$$

The RVM translation is

```
: SECTOR ( n n -- n.n.*.P )
   (: VALUE r VALUE c   :)
    r CORNER r CORNER 2 + ..
   c CORNER c CORNER 2 + .. PROD ;
```

This is a case where use of the stack commands would simplify matters a little, as

```
 r CORNER r CORNER
```

can become `r CORNER DUP` with the same effect.

The union of the row, column and sector sets produced above creates the Constraint Zone; the definition in the specification (being function-like in structure) also maps onto a straightforward operation in RVM.

```
: CZONE ( n n  -- n.n.*.P )
  (: VALUE r VALUE c :)
  r GENROW c GENCOL \/ r c SECTOR \/ ;
```

Now the set of digits not used in this constraint zone is required as the possible assignments to the square (the set of *available* numbers for the square). Construction of this set involves the complement of the relational image of the constraint zone, specified earlier as

$$available(r, c) = DIGIT - grid[czone(r, c)]$$

Converted for postfix, it maps exactly to RVM-Forth:

```
: AVAILABLE ( n n  -- n.P )
   (: VALUE r VALUE c :)
   DIGIT GRID r c CZONE IMAGE \ ;
```

Example output, using the square (3,2) with a sample puzzle loaded, would be

```
3 2  AVAILABLE .SET (enter)
 {1,2,5,8,9} ok
```

All that remains in the machine *czones* is to initialise the set *possible*, pairing each blank square with the set of its available values. This is the operation **init_possible** in the specification, with RB0 as follows.

```
init_possible $\hat{=}$
  VAR $S, s, V$ IN
    $S := SQUARE - \mathsf{dom}(grid);$
    $possible := \{s :\in S;\ V := available(row(s), col(s)) \diamond s \mapsto V\}$
  END
```

In this, S is the set of blank squares. As noted, RVM has operations to extract the elements of a pair whatever its type (they leave either a scalar value or an address on the stack); these are `FIRST` and `SECOND`, so they are used instead of *row* and *col*. We will also assume the existence of corresponding polymorphic functions in RB0, so the penultimate line in the above may be written

$$possible := \{s :\in S;\ V := available(first(s), second(s)) \diamond s \mapsto V\}$$

and we use these in subsequent extracts from the specification to replace $poss2()$ and the other definitions.

```
: INIT-POSSIBLE ( --  )
   (: :)  0 VALUE S 0 VALUE s 0 VALUE V
   XY XY PROD GRID DOM \ to S
   INT INT PROD INT POW PROD {
      <RUN
         S CHOICE to s
         s FIRST s SECOND AVAILABLE to V
         s V |->
       RUN> } to POSSIBLE ;
```

The translation of the PV semantics follows the pattern described in section 5.1.1 above.

5.4.1 Moves

We progress to filling in the grid in the game. The first requirement is to obtain the set of most constrained squares — one shortcut we have available stems from the fact that, internally, the sets in RVM are stored in order. So they have their smallest element as the first one, which can be picked out automatically (by calling `ELEMENT` on it). The size of this corresponds to ***n*** in the RB0 below, and the call to *mcn* is replaced accordingly in the code which follows.

```
$mcset \longleftarrow$ **getsquares** $\hat{=}$
**VAR** $x, n$ **IN**
   $n := mcn(possible)$;
   $mcset := \{x :\in possible;\ \mathsf{card}(second(x)) = n \Longrightarrow \diamond\ x\}$
**END**
```

The code now uses the shortcut to find the size of the smallest set, and then builds the required subset of *possible*. This is left on the stack, so a separate assignment to an output variable *mcset* is not required.

```
: GETSQUARES ( -- n.n.*.n.P.*.P)
    (:  :)  0 VALUE n 0 VALUE x
   POSSIBLE RAN ELEMENT CARD to n
   INT INT PROD INT POW PROD {
      <RUN
         POSSIBLE CHOICE to x
         x SECOND CARD n = --> x
      RUN>  }  ;
```

From the set produced by this, the operation **nextup** chooses (non-reversibly) one square/ values pair, and subtracts it from *possible*. The RB0 is

```
$nx \longleftarrow$ **nextup** $(mset)$  $\hat{=}$
**VAR** $x$ **IN**
  $x :=$ choice$(mset)$;
  $possible := possible - \{x\}$;
  $nx := x$
**END;**
```

The code leaves `x` directly on the stack.

```
: NEXTUP ( n.n.*.n.P.*.P -- n.n.*.n.P.*   )
     (: VALUE mset :)   0 VALUE  x
     mset  PCHOICE to x
     POSSIBLE x SUBTRACT-ELEMENT to POSSIBLE  x  ;
```

The word `PCHOICE` in this context works as a non-reversible choice. It is in fact a probabilistic choice, with a number of other capabilities, and an accompanying reversible version `PCHOICE_`. We posited a function `choice` to represent this in RB0, which is used in the version above.

Now one value must be chosen *reversibly* from the set to actually assign to the square; the specification operation does this with its argument *nx* and one local variable, *n*.

```
**assign** $(nx)$  $\hat{=}$
**VAR** $n$ **IN**
  $n :\in second(nx)$;
  $assigned := first(nx) \mapsto n$;
  $grid := grid \cup \{assigned\}$
**END;**
```

A direct translation of this would be as follows

```
: ASSIGN ( n.n.*.n.P.* --  )
   (: VALUE nx  :)   0 VALUE n
   nx SECOND  CHOICE to n
   nx FIRST n  |->P,I  to ASSIGNED ( Needs type for constructor )
   GRID ASSIGNED ADD-ELEMENT to GRID ;
```

Optionally, a `0LEAVE` can be added at the end. Unfortunately, neither of these work when the program reverses; the problem is to do with a trade-off between efficiency and general applicability in the part of the RVM mechanism which deals with frame restoration on reverse operation. The reversal from a subsequent point will result in the `CHOICE` being made again, so the stack frame for this whole operation should really be restored. But unless specified, it may not be, as it could in general prove expensive computationally.

As documented in the RVM manual [27], the problem occurs because the choice occurs after the parameter *nx* has been instantiated, however not all of the stack is restored; probably only the top two items in the above example. So `n` and the set created by `nx SECOND` are restored, but not the value of `nx` itself. The call to `FIRST` therefore attempts to access memory location 0, and a segmentation fault invariably occurs.

There are a number of ways around this, the simplest being to add `0LEAVE_` to the end of the definition. This is the reversible version of `0LEAVE`, and ensures stack restoration. If concise local syntax is being used, i.e. `(: nx :)`, the word `FRAME-RESTORE` has the same effect.

A further possibility is to add a local variable to store the `FIRST` of `nx` (the square), which is not in the specification. The code is then

```
: ASSIGN ( n.n.*.n.P.* --  )
   (: VALUE nx :)   0 VALUE n
   nx FIRST VALUE SQ   ( New local variable )
   nx SECOND CHOICE to n
   SQ n |->P,I  to ASSIGNED
   GRID ASSIGNED ADD-ELEMENT to GRID ;
```

So in this case, a simplistic default translation using only non-optional code fails. The compiler could be made smart enough to spot the presence of a potential reversibility problem and adjust the stack frame restoration behaviour accordingly, perhaps if more than two stack items were in play. Then it could ensure that `FRAME-RESTORE` is added.

Sometimes potential optimisations involve a substantial departure from the RB0 version. For instance in `ASSIGN`, the following also achieves the required reversible outcome quite neatly without the variable `n`, or any additional restoration instructions.

```
: ASSIGN ( n.n.*.n.P.* --  )
   (: VALUE nx  :)
   nx FIRST nx SECOND  CHOICE  |->P,I  to ASSIGNED
   GRID ASSIGNED ADD-ELEMENT to GRID ;
```

But there is no way to specify this *in this form.* An ad hoc workaround might be possible using a function to encapsulate the non-deterministic choice from a set, however this is surprisingly involved — requiring a separate version for each type of set *and* running into problems with assigning from the empty set.

Such a facility would possibly be useful in RB0, to use the set choice anonymously without requiring an explicit assignment (as a direct argument to another operation or function), as this translates simply into RVM. However, like the pair extraction functions *first* and *second*, it would have to be polymorphic in nature such that the output would transparently be of the same type as the elements of the set without further user intervention. We have already used such a construct of a non-reversible choice in the substitution `choice`, corresponding to one of the uses of the RVM `PCHOICE`.

5.4.2 The Update procedure

As we saw in the specification, the update needs to remove the value just assigned from the blank squares which are no longer entitled to it. So the first thing is to generate the subset of `POSSIBLE` which needs updating. This is done by finding the constraint zone of the square with `CZONE`, and performing a domain restriction on `POSSIBLE`. If the resulting set is not empty — which would mean that all the squares in the zone have been filled — the operation now builds a new set of this type to update `POSSIBLE`. The crucial part is the test for the empty set in the range, which would mean that at least one square has no more possible values, completion is infeasible, and the program must reverse. Otherwise, `POSSIBLE` is updated by functional override.

```
**update_possible** $\hat{=}$
**VAR** $x, U, upd$ **IN**
  $U := czone(row(first(assigned)), col(first(assigned))) \lhd possible$;
  **IF** $U \neq \varnothing$ **THEN**
    $upd := \{x :\in U \diamond first(x) \mapsto second(x) - \{second(assigned)\}\}$;
    $\varnothing \notin \mathrm{ran}(upd) \Longrightarrow possible := possible \Lleftarrow upd$
  **END**
**END**
```

In the code below, the word `<|` denotes domain restriction, while the word `<+` denotes functional override. `ASSIGNED` is a variable holding the last assigned value, and `UNPAIR` leaves the first and second elements in that order on the stack (a minor abbreviation for some repetitive code). We use `SUBTRACT-ELEMENT` rather than building a set from the value and using set subtraction.

```
: UPDATE-POSSIBLE (  -- )
   (:  :)  0 VALUE U  0 VALUE upd  0 VALUE x
   ASSIGNED FIRST UNPAIR CZONE POSSIBLE <| to U
   U ?{} NOT IF
      INT INT PROD INT POW PROD { <RUN
         U CHOICE to x
         x FIRST  x SECOND ASSIGNED SECOND SUBTRACT-ELEMENT |->
       RUN> } to upd
       INT { } upd RAN IN NOT   -->
           POSSIBLE upd <+ to POSSIBLE THEN  ;
```

The remaining program deals with the initial puzzle loading and set-up (and the display interface), and with the loop that runs each step, or "move". Using the optimisation suggested on page 42, we have the code

```
: STEP  ( -- )
    GETSQUARES NEXTUP ASSIGN UPDATE-POSSIBLE  ;
```

```
: SOLVE ( -- n.n.*.n.*.P )
   BEGIN
      GRID CARD 81 <
   WHILE
      STEP
   REPEAT  ;
```

A complete listing with initialisation from a file, running and printing out code is included as Appendix C.

5.5 Reversible Random Number Generator

A subtlety that arose in the formalising of reversibility constructs is that of generating random numbers. Strictly speaking, if a sequence of instructions (program fragment) is executed again *after* reversing, we should require that the same random numbers be generated as on earlier runs. Then the conditions will be identical for each run, and we can formally posit an equivalence.

For instance, in the GSL identity

$$(S \,[]\, T)\,;\; U = (S\,;\; U)\,[]\,(T\,;\; U)$$

there is a choice between two programs, S and T, followed by the program U. Should U contain a call to a Random Number Generator (RNG), the result should be same in both instances of U in the right hand side. In a reversible scenario, this program could run two or more times, and — formally at least — it should be the same program each time.

However, most RNGs in programming languages have a “seed” value, from which they algorithmically produce a pseudo-random sequence of numbers. So the values on successive runs of U will not be the same. But it can be arranged in the implementation (via an interface of C code) that the RNG is attached to a state array, initialised by `initstate`, which can be used as a history stack for generated random numbers. Thus if a sequence of numbers occurs, reversing to any point in the sequence will repeat the subsequent numbers in the sequence.

In practice, although very brief to code, this is effectively unnecessary (one pseudo-random number being as good as another for any particular run). But it does allow the formal requirements to be achievable in principle. The code and sample runs are included as Appendix F.

5.6 Summary

We have described the RVM, mostly from the standpoint of the programming interface, with regard to the changes made to support reversibility and abstract data types; this addresses Objective 4. An example from chapter 2 was expanded and translated to compare the parameter-passing with a C translation, and show a new possibility for implementation.

Various smaller examples and the Sudoku program from chapter 4 were coded in RVM-Forth, as directly as possible from the RB0, anticipating the translation schemas to be introduced in chapters 6 and 7, and addressing the remainder of Objectives 2 and 3. The constructs of RVM-Forth are summarised in Appendix B.

A number of elements in the previous examples have included simple constant functions, translated into RVM-Forth on an ad hoc basis. There is a potential for more complex and powerful function structures to be used in this way, along with other less-used facilities in B; we examine these and their translations in the next chapter.

Chapter 6

A More Powerful Implementation Language

In this chapter we examine ways of extending the role of abstract constants which define functions in B, including the use of Prospective Value computations. These allow state to be used in a function without leaving any trace, with formal semantic support; we include two examples of this, a simple GCD function which uses a state variable, and a minimax routine which could use the game state without altering it. This would be one useful way in which state could be used in a function — to calculate the set of states reachable from the current state, so that this set could be subject to heuristic evaluation, for instance. A way to enable reliable translation of such programs into RVM is introduced along with translation schemas, and some implications for proof obligations are noted.

In section 6.2 we look at a feature of the RVM which allows lambda calculus techniques to be used beyond the current limited use at abstract level. In this way we can use functions as first-class objects, creating anonymous or named functions as required, and using the technique of closure. The bound variables of lambda constructs are implemented with RVM local variables, which also cover function variables and standard operation variables. **Note:** Some of the material in section 6.2 is based on an earlier treatment in a paper given at the 22nd Euroforth conference [17].

6.1 An extended functional language of concrete constants

There is a capability in B, in its abstract machines, for using functions defined as properties of constants. We have used a number of fairly simple examples in earlier case studies. These definitions are more akin to those found in functional languages, and their conceptual ancestor lambda calculus (to which we return later), rather than procedural specifications. They may also involve recursion, which is not catered for in B operations — such a function would become an operation with a loop in the implementation.

The purpose of such a function is not to change state in the machine; it will merely run and leave an output. So proof obligations concerning the preservation of the Invariant would not apply. However they cannot persist to the implementation stage, as constants must be made concrete — for instance numerical constants must be given an actual value. Whereas functions remain abstract constants unless they can be refined as arrays or sequences. The one proof obligation at the abstract level is that for existence; does a function with the stated properties exist?

One issue with the fact that functional constants are specified as a static set of properties is the difficulty of automating a translation to RVM code. There is usually more than one way of expressing a set of properties as a conjunction of predicates, and recursion requires a certain syntactic framework in RVM.

We propose, therefore, to augment the existing system of abstract functions with a restricted syntax for implementable functions in RB0, based on a functional language style, in which the abstract functions are re-expressed in a way which makes automated translation easier. This language will also allow Prospective Value expressions to be used, and thus potentially allow state variables to be referred to and altered inside a function, while leaving no trace on the state once the function has terminated.

The existence proof obligation of the original abstract function is now obviated by the fact that we have an actual function specified in a functional style; however the properties of the two would need to be linked in some way. The abstract version can be implemented in several ways, so this really specifies a family of functions; what would need to be shown is that the RB0 version is indeed a member of this family, that is, it shares the mathematical properties of the abstract function.

We begin with a simple example of the factorial function, which could be specified as

PROPERTIES

$fact \in \mathbb{N}_1 \rightarrow \mathbb{N}_1 \wedge$

$fact(0) = 1 \wedge fact(1) = 1 \wedge$

$n > 1 \Rightarrow fact(n) = n \times fact(n-1)$

One possible RVM operation to implement this is

```
: FACT  ( n -- n )
   DUP 1 > IF
   DUP 1- RECURSE * THEN ;
```

This does not intuitively map onto the specification; the use of `RECURSE` rather than the function name could be overcome, but the general structure of the specification does not lend itself to systematic conversion. For example, there is no explicit conditional in the specification, but rather a list of cases; also the properties could be listed in a different order. Since Forth is a procedural language, there is a paradigm shift from the specification to the code that would require human input to overcome.

The constant function would at present (if recursive) most likely be implemented as a loop; it can be referenced in a loop invariant in an implemented operation, but not directly called. However, were the function allowed to persist to the implementation stage, by being re-expressed in the RB0 expression language, we note that it has no effect on state while running, and simply outputs a value when done. It would only remain to be shown that the output would match that from the abstract specification, for a given input.

A more involved example is based on the Towers of Hanoi, a venerable and well-known problem. This lends itself to a recursive solution, as the problem breaks up into smaller versions of itself, winding up with a single disc being moved. Varying numbers of poles and discs can be accommodated, but we restrict ourselves to a simpler version with three poles. The larger the number of discs, the more levels of recursion will be required.

Having declared a constant *POLE*, which is the set $1..3$, we specify the properties of the constant *hanoi* as follows. The variables for the function are *source* and *dest*, the source pole and destination pole, and *discs*, the number of discs.

```
**PROPERTIES**
$\ldots \wedge$
$hanoi \in POLE \times POLE \times \mathbb{N}_1 \twoheadrightarrow \mathrm{seq}(POLE \times POLE) \wedge$
$\forall (source, dest, discs).(source \in POLE \wedge dest \in POLE \wedge discs \in \mathbb{N}_1 \Rightarrow$
  $(discs = 1 \Rightarrow (hanoi(source, dest, discs) = [source \mapsto dest]) \wedge$
  $discs > 1 \Rightarrow hanoi(source, dest, discs) =$
    $hanoi(source, 6 - (source + dest), discs - 1) ^\frown [source \mapsto dest]$
    $^\frown hanoi(6 - (source + dest), dest, discs - 1)))$
```

A way of aliasing the quantity $6 - (source + dest)$ (to *via*, for instance) would aid comprehensibility here; this represents the number of the "other" pole, the one that is not *source* or *dest*. In the actual code an ad hoc local variable or constant might be used to achieve this. We introduce a way of doing this in the RB0 version shortly.

The function above returns a sequence of "moves", which are pairs of poles; a shorthand for "move the top disc from the first pole and put it on the second". The function is called from within an operation as in the following, which checks the types of the arguments first and also that the source and destination poles are not the same (although it could be arranged for an empty sequence to be returned in this case).

```
$sol \longleftarrow$ **solve**$(src, dest, discs)\ \hat{=}$
  **PRE**   $src \in POLE \wedge dest \in POLE \wedge discs \in \mathbb{N}_1 \wedge src \neq dest$
  **THEN**   $sol := hanoi(src, dest, discs)$
  **END**
```

At some stage, the set $\mathbb{N}_1$ would be replaced by something more implementable, at least taking into account the limits on 32-bit integers.

The function can be coded in procedural form as below, using the `RECURSE` operation rather than the operation name. With three arguments, it makes more sense to use local variables, to keep track of what one is doing.

```
( Takes a SOURCE pole, a DEST pole and a number of DISCS )
: HANOI ( n n n --  n.n.*.seq )
   (: VALUE SOURCE VALUE DEST VALUE DISCS :)
   DISCS 1 = IF      ( one disc left to move )
      INT INT PROD [ SOURCE DEST |-> , ]
   ELSE
      SOURCE 6 SOURCE DEST + -  DISCS 1- RECURSE
      INT INT PROD [ SOURCE DEST |-> , ] ^
      6 SOURCE DEST + -  DEST DISCS 1- RECURSE ^
   THEN
   1LEAVE ;
```

A pair of sample runs follow. In the second, four discs are specified, so it requires twice as many moves to complete.

```
3 1 3 HANOI ok.
.SEQ  [(3,1),(3,2),(1,2),(3,1),(2,3),(2,1),(3,1)]ok
2 3 4 HANOI ok.
.SEQ [(2,1),(2,3),(1,3),(2,1),(3,2),(3,1),(2,1),(2,3),(1,3),
      (1,2),(3,2),(1,3),(2,1),(2,3),(1,3)]ok
```

Before considering other things that functional constants could be used for, we consider how the transition from the abstract specification to code might be eased for the above example and similar functions. Such constants are not usually refined as themselves, but rather absorbed into an operation (so proofs are required that the operation does exactly what the function did), and of course the expressiveness of AMN operation language is of a different sort to the expressiveness of the abstract specification. We consider how the constant properties might be re-expressed (rather than refined) in an implementation written in RB0, which would be more amenable to translation into code.

A possible bridge between the two might be modelled on the functional programming languages, such as Scheme, Lisp or Haskell, specifically in their ways of expressing recursive functions. The pattern followed depends on the typing policy of the language; it might be necessary to specify the type first or not (the latter is the case for the RVM), however this would be followed by a list of condition-action pairs. One (or more) of the actions may involve recursion, in which case at least one condition must include a terminating condition, or base case, which must always be reached. If the recursion occurs with a decreasing integer argument, for instance, the termination must involve a lower integer than the initial argument (or equal to it).

In the programming languages, it is usual to utilise the fact that the conditions or guards are evaluated in the order written; thus the terminating guard g_0 should come first, and one or more following guards can use the pattern

$$g_1 \vee g_2 \vee \cdots \vee g_n \vee \neg \left(\bigvee_{i=1}^{n} g_i \right) \tag{6.1}$$

The final condition could be introduced with a keyword such as `otherwise` (in Haskell), or may not even need specifying (as in RVM). Also any other conditions can use the fact that earlier cases have already been eliminated; in general the first case to result in *true* will be executed.

However in a specification we cannot rely on the order in this way. The guards must be exhaustive and mutually exclusive, such that exactly one will be chosen at any evaluation. There must be no undefined cases or overlapping ones. This will still produce perfectly executable code. We return to this with a more formalised requirement, and a possible syntactic alternative to specify an "otherwise" guard, in section 6.1.1.

The syntax in such an implementable constant-property specification (that is, in RB0) can make use of certain aspects of bunch syntax, in particular the bunch union (a comma) and the bunch guard ($\longrightarrow$).

We can illustrate this with a simpler example before returning to the Towers of Hanoi. A function *luc* to generate numbers in the Lucas sequence (similar to the Fibonacci sequence but beginning with 2,1 rather than 1,1) can be specified thus:

PROPERTIES
$\ldots \wedge$
$luc \in \mathbb{N} \rightarrow \mathbb{N} \wedge$
$luc(0) = 2 \wedge$
$luc(1) = 1 \wedge$
$\forall n.(n \in \mathbb{N} \wedge n \geq 2$
$\quad \Rightarrow luc(n) = luc(n-1) + luc(n-2))$

As the typing information has already been provided, a version of this in RB0 could use a more functional style of language to encapsulate the computational parts. This could then be more easily related to Forth, using the `CASE` construct. In the following, each condition guards an output, which is a bunch in this context. The commas are not punctuation, but actually gather the component bunches into a single resultant bunch.

$$\begin{aligned} luc(n) \ \ \widehat{=} & \\ & n = 0 \longrightarrow 2, \\ & n = 1 \longrightarrow 1, \\ & n > 1 \longrightarrow luc(n-1) + luc(n-2) \end{aligned} \tag{6.2}$$

The syntax defines the output of the operation as a union of guarded bunches. But since the conditions are exhaustive with respect to n being a natural number, and mutually exclusive, only one of these will actually be executed. We note that a closing symbol of some sort would be desirable if more than one function is to be defined in the relevant RB0 clause.

Now translating this to Forth looks much more possible, allowing for postfix conversion. The `CASE` construct would normally compare integers with a value on the stack to decide a course of action. Alternatively, it can use the word `?OF` to react directly to a boolean flag left by a test; if the test is true the action between `?OF` and `ENDOF` is executed. This latter is the more appropriate construction here; this gives us

```
: LUC ( n -- n )
   (: VALUE N :)
   CASE
      N 0 = ?OF 2 ENDOF
      N 1 = ?OF 1 ENDOF
      N 1 > ?OF  N 1- RECURSE N 2 - RECURSE + ENDOF
   ENDCASE 1LEAVE ;
```

The third condition could be omitted here; as long as the function conditions had been properly specified, it would simply be an "otherwise" case, and the line beginning `N 1 >` could be shortened to read

```
 N 1- RECURSE N 2 - RECURSE +
```

Further, in a situation with only two choices, this could be translated using an `IF-ELSE-THEN` construct (again some extra work for the translation engine, or perhaps this could be embedded in RB1).

Our first example of the factorial function *fact* on page 64 can be treated similarly to produce

$$\begin{aligned} &fact(n) \ \hat{=} \\ &\qquad n \leq 1 \longrightarrow 1, \\ &\qquad n > 1 \longrightarrow n \times fact(n-1) \end{aligned}$$

Whereupon the RVM produced from a translation (converted to `IF-ELSE` form) would be

```
: FACT  ( n -- n )
   (: VALUE N  :)
   N 1 <=
   IF 1
   ELSE N N 1- RECURSE *
   THEN ;
```

The RB0 version of *hanoi*, in the style of *fact* and *luc* ((6.2) above), would now have the following kind of structure.

$$\begin{aligned}
&\mathit{hanoi}(\mathit{source}, \mathit{dest}, \mathit{discs}) \;\hat{=}\\
&\qquad \mathit{discs} = 1 \longrightarrow [\mathit{source} \mapsto \mathit{dest}],\\
&\qquad \mathit{discs} > 1 \longrightarrow \mathit{hanoi}(\mathit{source}, 6 - (\mathit{source} + \mathit{dest}), \mathit{discs} - 1)\\
&\qquad\qquad\qquad\qquad \frown ([\mathit{source} \mapsto \mathit{dest}]\\
&\qquad\qquad\qquad\qquad \frown \mathit{hanoi}(6 - (\mathit{source} + \mathit{dest}), \mathit{dest}, \mathit{discs} - 1))
\end{aligned}$$

This is clearly easier to relate to the Forth version `HANOI` on page 66.

As mentioned earlier, a local variable could be used to alias awkward to read or multiply-used quantities such as $6 - (\mathit{source} + \mathit{dest})$. This could perhaps be rendered in RB0 using the syntax

$$\textbf{Let } x = F \textbf{ In } E \textbf{ End}$$

Analogues to this construct are found in many functional languages, with various syntaxes. The above is adapted from a similar AMN construction. In the example above, such a usage would yield

$$\begin{aligned}
&\mathit{hanoi}(\mathit{source}, \mathit{dest}, \mathit{discs}) \;\hat{=}\\
&\textbf{Let } \mathit{via} = 6 - (\mathit{source} + \mathit{dest}) \textbf{ In}\\
&\qquad \mathit{discs} = 1 \longrightarrow [\mathit{source} \mapsto \mathit{dest}],\\
&\qquad \mathit{discs} > 1 \longrightarrow \mathit{hanoi}(\mathit{source}, \mathit{via}, \mathit{discs} - 1)\\
&\qquad\qquad\qquad\qquad \frown ([\mathit{source} \mapsto \mathit{dest}]\\
&\qquad\qquad\qquad\qquad \frown \mathit{hanoi}(\mathit{via}, \mathit{dest}, \mathit{discs} - 1))\\
&\textbf{End}
\end{aligned}$$

Here, *via* is not a true variable, rather a syntactic substitution; for formal purposes it is merely expanded to its definition. In Forth, the second condition action of `HANOI` now becomes more comprehensibly rendered as

```
   (SCOPE 6 SOURCE DEST + - VALUE VIA ( local variable )
( ... )
   SOURCE VIA  DISCS 1- RECURSE
   INT INT PROD [ SOURCE DEST |-> , ] ^
   VIA DEST DISCS 1- RECURSE ^  SCOPE)
```

The `(SCOPE ... SCOPE)` delimiters would not be required in smaller functions, where the scope is the whole function in any case.

Another limitation of functions defined in B is that they cannot refer to state variables; were they able to change state, they would be liable to all the proof obligations entailed in such a capability. But we have seen cases where operations can be carried out reversibly on state, using the prospective value facilities. In these situations, no lasting change is applied to the state, and no proof obligations need be incurred; if a function could refer to state variables *in this way*, within a reversible construct, then it might be possible to use state and state-based operations along with a functional style. Naturally, *only* such a usage would be allowable, that is, any change is confined within a prospective value computation.

To show how such an enterprise might be achievable, we introduce the example of a greatest common divisor function, $\mathsf{gcd}(x, y)$, which would return the gcd of x and y. We can define this recursively in a function. Supposing it was desirable to use a state variable y directly in the machine, rather than passing its value in as an argument. The normal ways of implementing gcd will change the state of such a variable.

At specification level we can define a two-variable *gcd* function, plus a dummy function g, which takes one variable, as follows:

PROPERTIES
$gcd \in \mathbb{N} \times \mathbb{N} \to \mathbb{N} \wedge$
$\forall(u, v).(u \in \mathbb{N} \wedge v \in \mathbb{N} \Rightarrow$
$((u = v \Rightarrow gcd(u, v) = u) \wedge$
$(u > v \Rightarrow gcd(u, v) = gcd(u - v, v)) \wedge$
$(u < v \Rightarrow gcd(u, v) = gcd(u, v - u)))) \wedge$
$g \in \mathbb{N} \to \mathbb{N} \wedge \quad /*\ Dummy\ function\ */$
$\forall(x, y).(x \in \mathbb{N} \wedge y \in \mathbb{N} \Rightarrow g(x) = gcd(x, y))$

As it stands here, g has no connection with the machine variable y, which has been defined in the **VARIABLES** clause. We would redefine it in a further detailed definition for RB0, using the functional bunch style introduced above, with $S \diamond E$ for the prospective value computation involving y. This is appropriate in a bunch union context, as the output of a PV computation is defined as a bunch.

$$
\begin{aligned}
g(x) \ \hat{=}& \\
&x = y \longrightarrow x, \\
&x > y \longrightarrow g(x - y), \\
&x < y \longrightarrow (y := y - x \diamond g(x))
\end{aligned}
$$

The assignment $y := y - x$ will not actually have altered the value of y at the end of $g(x)$. We understand the term $y := y - x \diamond g(x)$ as representing the value $g(x)$ *would* have were it to be run after the assignment $y := y - x$ (a prospective value, in fact).

This translates directly into RVM, with the PV computation being implemented by a `<RUN ...  INT>` construct. This is not being used within a set construction (as only one value is expected to be returned), and so requires the type information to be supplied explicitly in the word `INT>`.

In the following code, the variable `Y` is global (i.e. machine level), and must be declared as reversible, in order that its original value be restored at the end of the operation. The final condition `X Y <` has also been omitted.

```
36 VALUE_ Y

: G ( n1 -- n2, output is gcd(n1,Y)  )
   (: VALUE X :)
   CASE
      X Y = ?OF X ENDOF
      X Y > ?OF X Y - RECURSE ENDOF
      <RUN
          Y X - to Y
          X RECURSE INT>
   ENDCASE 1LEAVE ;
```

As a application of using state in a functional setting, we present a Minimax function; the machine operations which update the state of the game are used only in a reversible context, and state can be accessed (read-only) directly. Minimax requires some heuristic to assign a numerical score to any game position, for instance a high score might advantage player A and low score player B.

In the following, *score* is the heuristic to compute the numerical score, the functions Smin and Smax respectively return the minimum and maximum of a set, the parameter n allows for a $2n$ move lookahead, and the machine operations Amove and Bmove non-deterministically play a move for players A and B. There will be different evaluation strategies for players A and B; that shown below is the one for player A to evaluate a provisional move.

$$\begin{aligned} &Aeval(n) \;\hat{=} \\ &\qquad n = 0 \longrightarrow score, \\ &\qquad n \neq 0 \longrightarrow \text{Smin}\,(\{\text{Bmove} \;\diamond\, \text{Smax}\,(\{\text{Amove} \;\diamond\, Aeval(n-1)\})\}) \end{aligned}$$

A translation of this, using `SETMIN` and `SETMAX` for Smin and Smax, is

```
: AEVAL ( n -- n )
   N 0=
   IF
      SCORE
   ELSE
      INT { <RUN
         BMOVE INT { <RUN
            AMOVE N 1- RECURSE RUN> } SETMAX
      RUN> } SETMIN
   THEN 1LEAVE ;
```

As in functional programming, *Aeval(n)* is an expression with a value, however it has (limited) access to state. Neither *Aeval* (nor the player B version) nor *score* need be passed the state of the game, and although Amove and Bmove change the state, they do so only reversibly in this context, and the minimax search is guaranteed to have no side-effects.

6.1.1 Translation Schema for Functions

If we can standardise the syntax for the functions which have been redefined in an RB0 implementation (there is as yet no specialised clause for this), from an initial set of conditions for the abstract function, we can define translations schemas for the most common cases. We assume the type information has been supplied in the original Properties clause. In the following, the notation $[\![E]\!]$ indicates the result of translating the RB0 expression E into RVM, and the arrow $\leadsto$ means "becomes".

The simplest function is without guards, in this form

$$[\![\, \mathit{fname}(x_1, x_2, \ldots, x_r) \;\hat{=}\; E \,]\!] \;\leadsto$$

```
: $\mathit{fname}$ (: VALUE_ $x_1$ VALUE_ $x_2$ $\cdots$ VALUE_ $x_r$ :)
$[\![\, E \,]\!]$ ;
```

There is a possibly empty list of r arguments x_i, and the expression E is taken to leave one value on the stack.

More complex definitions will involve conditionals, in which only one action must be executed (which may include recursion). In these we have a collection of guards $g_1, g_2, \ldots g_n$ and corresponding actions or expressions $E_1, E_2, \ldots E_n$. For the guards we have two structural preconditions to ensure that at least one and only one action is executed.

1. Exhaustive: at least one is true, i.e.

$$\bigvee_{i=1}^{n} g_i = \mathit{true}$$

2. Mutually exclusive: no more than one can be true, i.e.

$$g_j \wedge g_k = \mathit{false}, \qquad j \neq k \text{ and } j, k \in 1, 2, \ldots, n$$

A possible optimisation would be for the case of two guards only, where an `IF-ELSE` construction can be used rather than compiling a `CASE` construct. This requires that the translation engine be able to recognise such an occurrence; assuming it can, this would yield

$$\begin{aligned}&[\![\, \mathit{fname}(x_1, x_2, \ldots, x_r) \;\hat{=}\\ &\quad g_1 \longrightarrow E_1,\\ &\quad g_2 \longrightarrow E_2]\!] \;\leadsto\end{aligned}$$

```
: $\mathit{fname}$ (: VALUE_ $x_1$ VALUE_ $x_2$ $\cdots$ VALUE_ $x_r$ :)
  $[\![\, g_1 \,]\!]$ IF $[\![\, E_1 \,]\!]$
   ELSE $[\![\, E_2 \,]\!]$ THEN  ;
```

The two guards here are complementary, such that $g_1 \Rightarrow \neg g_2$ and $g_2 \Rightarrow \neg g_1$, from the requirement above.

With more than two guards, the general form will be

$$\begin{aligned}
&[\![\, fname(x_1, x_2, \ldots, x_r) \ \hat{=} \\
&\quad g_1 \longrightarrow E_1, \\
&\quad g_2 \longrightarrow E_2, \\
&\quad \vdots \\
&\quad g_n \longrightarrow E_n \ ]\!] \ \leadsto
\end{aligned}$$

```
: fname (: VALUE_  x1 VALUE_  x2 ··· VALUE_  xr :)
  CASE
    [[ g1 ]] ?OF [[ E1 ]] ENDOF
    [[ g2 ]] ?OF [[ E2 ]] ENDOF
    ⋮
    [[ gn ]] ?OF [[ En ]] ENDOF
  ENDCASE ;
```

We mentioned earlier that the RVM CASE statement can have an empty final condition, which is implicitly an “otherwise” condition, meaning “not any of the preceding conditions”, as shown in expression 6.1 on page 67. A possible keyword such as *otherwise* could then be employed as a syntactic shortcut, with this precise definition: given a set of conditions $g_1, g_2, \ldots g_m$ which have been already used, then

$$\text{otherwise} \ \hat{=} \ \neg(g_1 \vee g_2 \vee \ldots \vee g_m)$$

The RVM’s CASE statement will thus treat this as its final unstated guard.

Using this, the two-guard function would now appear as

$$\begin{aligned}
&[\![\, fname(x_1, x_2, \ldots, x_r) \ \hat{=} \\
&\quad g_1 \longrightarrow E_1, \\
&\quad \text{otherwise} \longrightarrow E_2]\!] \ \leadsto
\end{aligned}$$

```
: fname (: VALUE_  x1 VALUE_  x2 ··· VALUE_  xr :)
  [[ g1 ]] IF [[ E1 ]]
   ELSE [[ E2 ]] THEN  ;
```

While with more than two guards, the general form will be

$$\begin{aligned}
&[\![\, fname(x_1, x_2, \ldots, x_r) \ \hat{=} \\
&\quad g_1 \longrightarrow E_1, \\
&\quad g_2 \longrightarrow E_2, \\
&\quad \vdots \\
&\quad g_{n-1} \longrightarrow E_{n-1}, \\
&\quad \text{otherwise} \longrightarrow E_n \ ]\!] \ \leadsto
\end{aligned}$$

```
: fname (: VALUE_  $x_1$ VALUE_  $x_2$ $\cdots$ VALUE_  $x_r$ :)
  CASE
    $[\![ g_1 ]\!]$ ?OF $[\![ E_1 ]\!]$ ENDOF
    $[\![ g_2 ]\!]$ ?OF $[\![ E_2 ]\!]$ ENDOF
    ⋮
    $[\![ g_{n-1} ]\!]$ ?OF $[\![ E_{n-1} ]\!]$ ENDOF
    $[\![ E_n ]\!]$
  ENDCASE ;
```

We also need the translation of the syntax for local aliasing introduced in the Towers of Hanoi example. This has the form

$[\![$**Let** $x = F$ **In** E **End**$]\!]$ $\rightsquigarrow$

`(SCOPE` $[\![F]\!]$ `VALUE x` $[\![E]\!]$ `SCOPE)`

There are also certain other features of the functional style which are not currently catered for, and might prove useful, which we look at in the next section.

6.2 Lambda expressions

The general form for a lambda expression is:

λ <name> . <body>

In the above, <name> is the parameter or *bound variable* in the function — lambda functions have a single parameter only. The process of substituting an argument for the parameter in an application is known as β-reduction (shown as $\xrightarrow{\beta}$), which substitutes occurrences of the bound variable in the body with the argument expression. Once thus instantiated, the variable cannot change value.

The <body> itself can be anything from a simple operation on the variable to other nested functions, including embedded function applications.

The simplest example would be the identity function:

$\lambda x.x$

which returns its argument; so

$(\lambda x.x)\ a \ \xrightarrow{\beta}\ a$

For further examples, we allow the use of arithmetic operations[1]. So the functions:

$\lambda x.x + 1$

$\lambda x.x \times x$

[1]Neither these nor numbers are initially present in the basic lambda calculus, which is concerned with representing them in terms of more primitive substitution patterns.

would respectively increment their argument by one, and square it.

$$(\lambda x.x \times x)\ 5 \ \xrightarrow{\beta}\ 5 \times 5 = 25 \tag{6.3}$$

These examples all have a single bound variable in the body; variables in the body which are not bound by the immediate lambda declaration are known as *free variables*; these make little sense computationally unless they are in turn bound by an enclosing function (or perhaps another kind of environment, in a non-pure lambda setting). In this example:

$$\lambda y.x - y \tag{6.4}$$

the variable x is free. It may, however be bound by an enclosing function whose *body* is (6.4):

$$\lambda x.(\lambda y.x - y)$$

This is the basic way to deal with two or more arguments in Lambda calculus (known as "currying"). The variable x has now become a local variable from an outer scope. We use an eager evaluation approach which requires it to be instantiated *before* the expression containing it is evaluated.

As regards application, parameters from left to right are substituted by arguments in the same direction, thus in an application of the above:

$$(\lambda x.(\lambda y.x - y))\ \ 10\ \ 3 \ \xrightarrow{\beta}\ (\lambda y.10 - y)\ \ 3 \tag{6.5}$$

The first reduction, substituting 10 for x, now returns a *function* which subtracts its argument from 10; here y would be substituted by 3 in the next reduction.

$$\xrightarrow{\beta}\ 10 - 3 = 7$$

An argument may well be another function. The expression:

$$\lambda f.(\lambda y.f\ y)$$

is a function that applies any given function f to any given argument y, for instance given the function $\lambda x.x + 1$ for f and the number 7 for y

$$\begin{aligned}
&(\lambda f.(\lambda y.f\ y))\ (\lambda x.x + 1)\ 7 \\
&\xrightarrow{\beta}\ (\lambda y.(\lambda x.x + 1)\ y)\ 7 \\
&\xrightarrow{\beta}\ (\lambda x.x + 1)\ 7 \\
&\xrightarrow{\beta}\ 7 + 1 = 8
\end{aligned}$$

Closures

The technique described above for dealing with functions of more than one variable (known as "currying") can be generalised where lambda expressions are used along with other types of expressions — in many programming languages, for instance.

The variable that was “free” in expression (6.4) could be a variable defined (and instantiated) in an outer environment. An environment in this context is a set of bindings, of variables to values.

We can illustrate this as in figure 6.1; one interpretation is that when *f(x)* is run in environment 2, the bindings of environment 1 are recalled and the value of w in this environment is used.

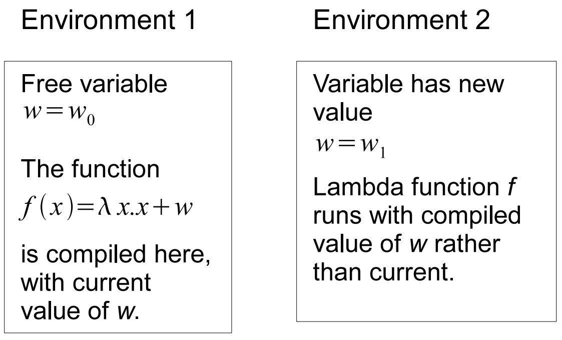
Environment 1
Environment 2
Free variable $w = w_0$
The function $f(x) = \lambda x.x + w$
is compiled here, with current value of *w*.
Variable has new value $w = w_1$
Lambda function *f* runs with compiled value of *w* rather than current.

FIGURE 6.1: Closure.

In practice, the necessary parts of the outer environment are compiled along with the function as a closed package, the whole being known as a closure. These are implementable in a variety of ways, which need not involve lambda expressions or capabilities. Indeed, the term is not used in the context of pure lambda calculus, as it reduces to simple substitution.

While closures are a useful technique, for instance in creating interface commands for instances of data structures (stacks, queues, and so forth), there is no way of specifying this sort of usage in B. For pure lambda calculus, the proof obligation overhead is very low, as they do not affect state, and so have no effect on the invariant.

6.2.1 Lambda Calculus in B

B provides some capability at the specification level only for lambda calculus; it can be used directly in operations or a constant can be defined as a lambda function[2]. To demonstrate the former usage, an example of an anonymous lambda function with nested lambdas is:

$$(\lambda z.(\lambda y.y + (\lambda x.x + y \times z)(y + z)))\ a\ b \tag{6.6}$$

After beta reduction this turns out to be a somewhat roundabout way of calculating

$$b + ((b + a) + ab)$$

We return to this example for the RVM implementation, as it has embedded function execution within the body of another function, but specifying it in a B

[2]We note that the documentation for such capability is virtually non-existent in our available tools.

machine operation looks like the following; note the types of each variable need to be supplied within the definition.

$$
\begin{array}{l}
o \longleftarrow \textbf{lambdemo}\ (a, b)\ \ \hat{=} \\
\quad \textbf{PRE}\ a \in \mathbb{N} \wedge b \in \mathbb{N} \\
\quad \textbf{THEN} \\
\qquad o := \lambda z.(z \in \mathbb{N} \mid \lambda y.(y \in \mathbb{N} \mid y + \lambda x.(x \in \mathbb{N} \mid x + y \times z)(y + z)))\ (a)\ (b) \\
\quad \textbf{END}
\end{array}
\tag{6.7}
$$

Alternatively, a constant can be defined as a lambda function in the following way. In this example, the function lf takes a function f as an argument and returns a function which applies that function to its own argument. In the unadorned lambda notation this is

$$\lambda f.(\lambda y.(f(y)))$$

For the B specification, type information is supplied to describe lf itself, and internally for each variable.

$$
\begin{array}{l}
\textbf{PROPERTIES} \\
\quad lf \in \mathbb{N} \rightarrow \mathbb{N} \rightarrow (\mathbb{N} \rightarrow \mathbb{N}) \wedge \\
\quad lf = \lambda f.(f \in \mathbb{N} \rightarrow \mathbb{N} \mid \lambda y.(y \in \mathbb{N} \mid f(y)))
\end{array}
$$

This can then be wrapped in an operation which takes the function g and a suitable argument a, and applies $g(a)$.

$$
\begin{array}{l}
o \longleftarrow \textbf{lambdemo2}\ (g, a)\ \ \hat{=} \\
\quad \textbf{PRE}\ g \in \mathbb{N} \rightarrow \mathbb{N} \wedge a \in \mathbb{N} \\
\quad \textbf{THEN} \\
\qquad o := lf\ (g)\ (a) \\
\quad \textbf{END}
\end{array}
$$

6.2.2 A Postfix Notation for Lambda Calculus

We introduce this notation purely for analysis, as a structural bridge between a B specification (or its RB0 version) and the RVM code. It would be unrealistic to expect a user specification or implementation to use it.

Function application in Lambda notation is usually shown by a bracketed function followed by an argument. Postfix will require the argument to appear first. Conventional Lambda notation uses brackets to delimit the scope of a bound variable, and for that we will use a specific `endλ` symbol. Finally conventional Lambda notation uses brackets to control when a function is applied so it can be taken from the stack, followed by a definition body or a symbol (defined earlier) standing for it. However, this would not be automatically applied; an additional symbol is required analogous to Forth's `EXECUTE`. The symbol we use for this is a tick $\boxed{\,'\,}$. This usage has a history going back at least as far as Principia Mathematica, where the application of function f to an argument x is written as $f'x$.

We can now present some examples from the earlier section in an abstract postfix notation, with the actual RVM-Forth code in the next section. We use the notation *infix* $\rightsquigarrow$ *postfix* to show how the infix and postfix forms correspond. Taking example (6.3), we have:

$$(\lambda x.x \times x)\ 5 \quad \rightsquigarrow \quad 5\ \lambda x.x\ x\ \times\ \mathsf{end}\lambda\ '$$

The λ and $\mathsf{end}\lambda$ keywords delimit the anonymous definition; at this point it could be assigned to a suitable variable, or applied, or left on the stack (in the RVM code this becomes an *execution token*, an address for the relevant executable code). The tick ensures application. Instantiation of the variable x by 5 (beta reduction) now yields the postfix expression “5 5 ×”, interpreted as usual.

The nested definition example (6.5), runs thus:

$$(\lambda x.(\lambda y.x - y))\ 10\ 3 \quad \rightsquigarrow$$
$$3\ 10\ \lambda x.\lambda y.x\ y\ -\ \mathsf{end}\lambda\ \ \mathsf{end}\lambda\ '\ '$$

Note that the arguments follow a stack order, first at the top. Above that is the function, however, with outer and inner variables as yet uninstantiated. So initially there are three items on the stack, as here:

$$\lambda x.\lambda y.x\ y\ -\ \mathsf{end}\lambda\ \ \mathsf{end}\lambda$$
$$10$$
$$3$$

The remaining ticks are outside the definition, and are executed as encountered. The return value of the outer function will be the inner function with its unbound variables bound.

The first tick will execute the outer definition (λx), which will instantiate x (anywhere within scope) to the value 10 from the top of the stack:

$$\xrightarrow{\beta}\ 3\ \lambda y.10\ y\ -\ \mathsf{end}\lambda\ '$$

This leaves 3 and the inner function on the stack — now with its x bound to a constant. The remaining tick executes this, instantiating y to 3, and returning “10 3 −”.

A possible use for this would be as an intermediate stage in the translation process, converting the lambda syntax to its postfix form, and perhaps including type information. This could be used in the context of a 2-pass compiler as described in Appendix G.

6.2.3 RVM-Forth Implementation

Local Variables and Lambda Parameters

RVM-Forth already has a facility for local variables, with the syntax :

```
: <opname> … (: VALUE <varname> … :) … nLEAVE … ;
```

The value for the local is taken from the stack — it can be an argument to the operation, or supplied by an expression within it. The keyword `nLEAVE` (where `n` can be 0, 1, 2, or 3) specifies the number of values left on the stack after the local environment goes out of scope.

Additional locals may be declared between the `:)` and the `nLEAVE`; they will be initialised from the top of the stack, so suitable values should be found there.

For the lambda implementation this format is used to represent the parameter for the lambda expression, instantiated from the stack.

As a matter of style, it might be noted that we use the same word `VALUE` for both global and local variables, with the latter version being defined in a `COMPILER` wordlist which is only searched when in Compile mode.

The Forth uses two keywords for the λ symbol itself. The word `:LAMBDA` opens a definition at the outer level, that is, the system enters compile mode. The corresponding endλ to this is `ENDLAM;`. The basic form is:

```
:LAMBDA (: VALUE <name> :) <body> 1LEAVE ENDLAM;
```

So far this is just an alternative syntax for the Forth Standard `:NONAME`. However, lambda definitions may also appear within compiled code, where they are delimited with the `LAMBDA` and `ENDLAM`. They may be nested to any depth.

The simple example (6.3) from page 75 translates from the abstract postfix notation as:

$$5\ \lambda x.x\ x\ \times\ \mathsf{end}\lambda\ '\ \rightsquigarrow$$

```
5 :LAMBDA (: VALUE X :) X X * 1LEAVE ENDLAM; EXECUTE
```

When evaluated this will leave 25 on the stack.

The example where a function forms part of the body, (6.5) on page 75, provides an illustration of binding from outside the function itself:

$$3\ 10\ \lambda x.\lambda y.x\ y\ -\ \mathsf{end}\lambda\ \mathsf{end}\lambda\ ''\ \rightsquigarrow$$

```
3 10
:LAMBDA (: VALUE X :)
   LAMBDA (: VALUE Y :)
      X Y - 1LEAVE
   ENDLAM 1LEAVE
ENDLAM; EXECUTE EXECUTE
```

The fact that `X` is free in the inner lambda is unremarkable in a straightforward execution such as this. However it is possible to name the inner function, using the keyword `OP`. This picks up the name from the input stream and assigns it to the execution token at the top of the stack, creating a global named operation. Instead of the final line in the above code, we could, for instance, have

```
  ... ENDLAM; EXECUTE OP MINUS
```

This gives us a named function which seems to refer to a variable `X` declared in a now-defunct scope and relating to a stack frame which no longer exists.

What has happened is that the evaluation of the outer `:LAMBDA` has instantiated the variable `X` to 10, so that when the inner lambda, i.e. the code

```
 ... LAMBDA (: VALUE Y :) X Y - 1LEAVE
```

is evaluated, `X` already has a value, and this is what is copied in place of *X* within the inner lambda definition.

A final example (specified as (6.7) on page 77 above) demonstrates embedded function execution within the body of another function. The standard infix lambda expression

$$(\lambda z.(\lambda y.y + (\lambda x.x + y \times z)\ (y + z)))\ 3\ 4$$

contains an embedded function application inside the λ*y* definition:

$$(\lambda x.x + y \times z)\ (y + z)$$

in which the *y* and *z* will have been substituted by arguments by the time this is evaluated. The expression as a whole converts to postfix lambda notation as

$$4\ 3\ \lambda z.\lambda y.y\ y\ z + \lambda x.x\ y\ z \times + \texttt{end}\lambda\ \ '+\ \texttt{end}\lambda\ \ \texttt{end}\lambda\ ''$$

The second argument to the final "+" is provided by the *result* of the inner function application

In RVM-Forth this becomes the following code.

```
4 3 :LAMBDA (: VALUE Z :)
  LAMBDA (: VALUE Y :)
    Y Y Z +
    LAMBDA (: VALUE X :)
      X Y Z * +
    1LEAVE ENDLAM EXECUTE  ( embedded function application)
    + 1LEAVE ENDLAM
  1LEAVE ENDLAM;
EXECUTE EXECUTE
```

The evaluation can be calculated "by hand" as follows:

"⟶ substituting 3 for `Z`"

```
4  LAMBDA (: VALUE Y :)
    Y Y 3 +
    LAMBDA (: VALUE X :)
      X Y 3 * +
    1LEAVE ENDLAM EXECUTE
    + 1LEAVE ENDLAM
EXECUTE
```

"⟶ substituting 4 for `Y`"

```
4 4 3 +
LAMBDA (: VALUE X :)
  X 4 3 * +
1LEAVE ENDLAM EXECUTE
+
```

"⟶ substituting 7 for `X`; definition immediately compiled and executed, which evaluates
`7 4 3 * +` as second argument to final `+`"

```
4  19  +
```

Leaving 23.

6.3 Summary

In this chapter we have developed the idea of a functional sublanguage in Reversible B. The functions are initially specified as abstract constants, but refined to concrete constants in a standardised format, with notation drawn from functional programming languages and bunch theory. A translation system for these was described demonstrating the appropriate RVM constructions, and thus the foregoing satisfies Objective 5.

Also we explored the possibility of an increased use of Lambda functions and expressions, with the corresponding RVM implementation for these structures and a way of approaching the translation from RB0 into RVM. This section addresses Objective 6.

In the next chapter we turn our attention to the main computational area in B, the operations and their translation.

Chapter 7

Translation Schemas for Operations

In this chapter, we consider a version of the B0 implementation level language (hereinafter RB0) with certain additions to the existing syntax. The translations are to RVM code; an initial obstacle is the conversion from infix notation to postfix, which we deal with first, moving on to more general expressions.

We translate from the symbolic form below; a machine-readable form of the symbolic language would need to be finalised, based on the ASCII character set (perhaps UTF-8 with an eye to Unicode compatibility and later expansion to include actual symbols). Symbols in existing B have an ASCII translation already, and in some cases these match the RVM operation names; as the translation would be run on a machine-readable file there would be no need to translate these in the actual process (we list these in the first subsection).

There is also the question of whether to base the translations on AMN — the "syntactic sugar" of Generalised Substitution Language, or RB0 itself with its extensions to GSL. We supply both versions where appropriate, as the exact status and extent of AMN with respect to RB0 or RB1 is not yet finalised.

The translations presented may not be exhaustive, but cover all of the more common constructs, and in addition the syntax for implementable function definitions, and operation definition/invocation using the stack to pass parameters is included. All of the constructs considered have implementations in RVM Forth.

Notation

Given an RB0 or RB1 formula F we use $[\![F]\!]$ to represent the result of translating F into RVM; we assume the necessary space-separation of resulting tokens is handled by the translation. We use `FIXED` font to represent the RVM code, and mathematical italic to represent variable parts of formulas, including operation names. We assume such names would be transliterated into RVM, which could potentially cause name clashes within the Forth environment. But this problem is separable from that of translation; so we do not consider the solutions to any such problems (for instance the use of word-lists in Forth).

The symbol $\rightsquigarrow$ is used to represent the translation process, the left-hand side "becomes" the right-hand side. Within formulas such as

$$y \longleftarrow Op(x)$$
$$f(E)$$
$$\text{VAR } x \text{ IN } S \text{ END}$$

the symbols x, y, and E are understood as standing for lists of variables or expressions (where appropriate), with respective components

$$x_1, x_2, \ldots, x_n$$
$$y_1, y_2, \ldots, y_m$$
$$E_1, E_2, \ldots, E_n$$

We write the list out explicitly when attention is being drawn to what happens to the components. In B the possibility exists for a formula such as $x := E$ to stand for a multiple assignment of the components of E to the components of x. However this is not used in our treatment, for reasons explained in section 7.5. Ellipses in either side of the translation (...) are meta-notation, and not literal parts of B or code.

7.1 Infix Binary Symbols

The general pattern for translating infix binary symbols is (where $\circ$ is a non-specific infix operation)

$$[\![F_1 \circ F_2]\!] \leadsto [\![F_1]\!] \, [\![F_2]\!] \, [\![\circ]\!]$$

Note that the operands remain in the same order, but the operator is moved to the end.

7.1.1 Binary Symbols which match existing ASCII versions

These are listed in table 7.1 below.

7.1.2 Other Binary Symbols

$[\![\wedge]\!] \leadsto$ `AND`
$[\![\in]\!] \leadsto$ `IN`
$[\![\notin]\!] \leadsto$ `IN NOT`
$[\![\neq]\!] \leadsto$ `<>`
$[\![\mathsf{mod}]\!] \leadsto$ `MOD`
$[\![\vee]\!] \leadsto$ `OR`

Symbol	**Translation**	**Symbol**	**Translation**
$\times$	`*`	$\Rightarrow$	`=>`
x^y (as `x**y`)	`**`	$⩤$	`<<|`
$+$	`+`	$\lhd$	`<|`
$-$	`-`	$\leq$	`<=`
..	`..`	$>$	`>`
/	`/`	$\geq$	`>=`
$\cap$	`/\`	$<$	`<`
$\uparrow$	`/|\`	$\cup$	`\/`
$\downarrow$	`\\|/`	$\frown$	`^`
$\lhd\!\!\!+$	`<+`	$\rhd$	`|>`
$\subseteq$	`<:`	$⩥$	`|>>`
$\subset$	`<<:`		

TABLE 7.1: Binary Symbols to ASCII

7.2 Translation of Equality

The $\boxed{=}$ symbol in RVM is not fully polymorphic, so in certain cases specialised versions of equality are required in the translation. These are required when we are comparing data structures such as strings, sets/ sequences, or pairs. Strings must have the same characters in each position, while the other structures must be compared recursively to ascertain deep equality.

Let F_1, F_2 be two expressions of the same type as specified in the following.
Integers:
$\llbracket F_1 = F_2 \rrbracket \rightsquigarrow \llbracket F_1 \rrbracket \llbracket F_2 \rrbracket$ `=`
Strings:
$\llbracket F_1 = F_2 \rrbracket \rightsquigarrow \llbracket F_1 \rrbracket \llbracket F_2 \rrbracket$ `STRING=`
Sets:
$\llbracket F_1 = F_2 \rrbracket \rightsquigarrow \llbracket F_1 \rrbracket \llbracket F_2 \rrbracket$ `SET=`
Pairs:
$\llbracket F_1 = F_2 \rrbracket \rightsquigarrow \llbracket F_1 \rrbracket \llbracket F_2 \rrbracket$ `PAIR=`

7.3 Prefix and Postfix Symbols

Most of these are function applications, whose arguments require brackets. The general pattern is

$$\llbracket f(E) \rrbracket \rightsquigarrow \llbracket E \rrbracket \llbracket f \rrbracket$$

Then we have

$[\![\text{card}]\!]$	$\leadsto$	`CARD`	$[\![\text{max}]\!]$	$\leadsto$	`MAX`
$[\![\text{dom}]\!]$	$\leadsto$	`DOM`	$[\![\text{min}]\!]$	$\leadsto$	`MIN`
$[\![\text{front}]\!]$	$\leadsto$	`FRONT`	$[\![\text{ran}]\!]$	$\leadsto$	`RAN`
$[\![\text{head}]\!]$	$\leadsto$	`HEAD`	$[\![\text{size}]\!]$	$\leadsto$	`CARD`
$[\![\text{last}]\!]$	$\leadsto$	`LAST`	$[\![\text{succ}]\!]$	$\leadsto$	`1+`
$[\![\text{tail}]\!]$	$\leadsto$	`TAIL`	$[\![\text{pred}]\!]$	$\leadsto$	`1-`

Unary minus differs in that it works like an operator, and requires no brackets in B.

$[\![-E]\!] \leadsto [\![E]\!]$ `NEGATE`

There is one postfix symbol in reasonably common use, the relational inverse.

$[\![R^{-1}]\!] \leadsto [\![R]\!]$ `~`

7.4 Declaration of Variables

For RB0 scalar variables, listed in the VARIABLES clause, we have

$[\![v_1, v_2, \ldots, v_n]\!] \leadsto$ `NULL VALUE_` v_1 `NULL VALUE_` v_2 ... `NULL VALUE_` v_n

These examples have been declared as reversible variables, which will have assignments undone during reverse computation. The alternative defining word `VALUE` can be used if this is not required, although signalling the choice is not expressible in B. The choice might be guided by a special comment, at least initially, or RB1 may be a better vehicle for such information. A more ambitious option would be semantic analysis of the RB0 to determine whether reversible variables would be required; detailed examination of this issue is beyond the scope of the thesis, but one case would be variables which, following any choice, are assigned to before being used in an expression. These would not need to be reversible.

Array variable *names* are declared in the same undifferentiated list in a B machine; the fact that they are arrays only becomes apparent in the INVARIANT clause, where they are defined as functions from the natural numbers to whatever type is in the array. It thus seems likely that consideration of the types of variables in the Invariant should precede their actual declaration in the RVM (this would also apply to sets and sequences). For instance an array variable v of n elements is declared as

n `VALUE-ARRAY_` v

or its non-reversible equivalent.

A more fruitful alternative would be to require some type information in the VARIABLES clause itself. This is in line with the practice in almost all common programming languages, and would allow — as variables in RVM need to be initialised on declaration — more meaningful initialisations than `NULL`, which is a typeless value, and can be applied to scalars, pairs, sets, and sequences. Conceivably information on required reversibility could be incorporated here, allowing for a choice to be made on whether `VALUE` or `VALUE_` is required.

7.5 Assignment

Individual scalar assignments are simple enough, but B does currently allow multiple assignments in one statement, in which case the order of values pushed and pulled from the stack must be taken into account. For a single scalar assignment we have

$$[\![x := E]\!] \quad \rightsquigarrow \quad [\![E]\!] \ \texttt{to}\ x$$

If multiple assignment of scalars were to continue with the current syntax, it has the variables in the same left-to-right order as the values which will be assigned to them; so one side must be reversed to accommodate the stack operation.

$$[\![x_1, x_2, \ldots, x_n := E_1, E_2, \ldots, E_n]\!] \quad \rightsquigarrow \tag{7.1}$$

$$[\![E_n]\!]\ [\![E_{n-1}]\!]\ \ldots\ [\![E_1]\!] \quad \texttt{to}\ x_1\ \texttt{to}\ x_2\ \ldots\ \texttt{to}\ x_n \tag{7.2}$$

However, there is a clash between this and another proposal involving bunches, for non-deterministic assignments to variables, (described in the next section). The problem is a clash in notation between this new form of assignment and B's existing multiple assignment as shown in expression (7.1). Here the right-hand side is not intended to be a bunch, but an ordered list of values. But there are other notations which can be used for this; at specification level, the parallel operator || (which does not occur in the current implementation-level language) yields

$$x_1 := E_1 \parallel x_2 := E_2 \parallel \ldots \parallel x_n := E_n$$

Parallel assignments are not allowed in B0, and would probably be refined to some form of sequential assignment. This could cause problems in a case such as

$$x := y \parallel y := x$$

Performing a sequential assignment $x := y;\ y := x$ would not have the desired effect of swapping the values of x and y. Letting the parallel form stand in RB0, and using the translation above, (7.2), allows all of the right-hand expressions to be evaluated first and put on the stack *before* any assignments are made. In the swapping example, where x_0 and y_0 are the initial values of x and y, this would yield the translation

$$x_0\ y_0\ \texttt{to}\ x\ \texttt{to}\ y$$

7.5.1 Assignment of Bunch Expressions

Starting with a simple example, we propose that

$$x := E_1, E_2$$

represent a non-deterministic assignment of expression E_1 or E_2 to x. The right-hand side is a bunch here, and the statement is equivalent to

$$x := E_1 \ [\,] \ x := E_2$$

As justification for this, we can show that the two are equivalent in PV semantics, as discussed in section 4.4 on page 29, as follows. In this, $\langle x/v \rangle$ means the preceding expression with x substituted for v, and the larger equals sign " $==$ " is of lower precedence than all the other operators, in particular the normal equals "=". So the top-level equivalences stand out from the equivalences in the individual expressions.

$$\begin{aligned} x := E_1, E_2 \diamond x &== x\langle E_1, E_2 / x \rangle \\ &== E_1, E_2 \end{aligned}$$

$$\begin{aligned} x := E_1 \ [\,] \ x := E_2 \diamond x &== x := E_1 \diamond x, x := E_2 \diamond x \\ &== E_1, E_2 \end{aligned}$$

Before moving on to the translation schemas for this construction, we must note that predicate transformer semantics does not support it; to see this we calculate $\mathsf{prd}(x := E_1, E_2)$ in both PV and predicate transformer forms (prd is discussed in section 2.1.7, page 12).

PV Semantics has (where x' is the value of x *after* the substitution)

$$\mathsf{prd}(S) \ \hat{=} \ x' : (S \diamond x)$$

So we obtain

$$\begin{aligned} \mathsf{prd}(x := E_1, E_2) &== x' : (x := E_1, E_2 \diamond x) \\ &== x' : E_1, E_2 \end{aligned} \tag{7.3}$$

The choice form gives the same result.

$$\begin{aligned} \mathsf{prd}(x := E_1 \ [\,] \ x := E_2) &== x' : (x := E_1 \ [\,] \ x := E_2 \diamond x) \\ &== x' : (x := E_1 \diamond x, x := E_2 \diamond x) \\ &== x' : E_1, E_2 \end{aligned}$$

The predicate transformer form uses the definition of prd as

$$\mathsf{prd}(S) \ \hat{=} \ \neg [S] x \neq x'$$

With this, we find that the choice form above gives an equivalent form in non-bunch notation:

$$\begin{aligned} \mathsf{prd}(x := E_1 \ [\,] \ x := E_2) &== \neg [x := E_1 \ [\,] \ x := E_2] x' \neq x \\ &== \neg ([x := E_1] x' \neq x \wedge [x := E_2] x' \neq x) \\ &== \neg [x := E_1] x' \neq x \vee \neg [x := E_2] x' \neq x \\ &== x' = E_1 \vee x' = E_2 \end{aligned}$$

The final line of this is operationally equivalent to the result in (7.3).

However, beginning with the PV form of the substitution and using the predicate transformer `prd`, we find

$$\begin{aligned}\mathsf{prd}(x := E_1, E_2) &\equiv \neg[x := E_1, E_2]x' \neq x \\ &\equiv \neg(x' \neq x\langle E_1, E_2/x\rangle) \\ &\equiv \neg(x' \neq E_1, E_2) \\ &\equiv x' = E_1, E_2\end{aligned}$$

The result in this case is *not* equivalent to that in (7.3), as this has x' ending up as a bunch; whereas it should be an elemental variable, with only a single value.

We therefore conclude that the use of bunch expressions in assignments in this way can be allowed *only* at the cost of remaining strictly within PV semantics.

The generalised translation for the non-deterministic assignment itself is

$$[\![\, x := E_1, E_2, \ldots, E_n \,]\!] \;\rightsquigarrow$$

$$\texttt{<CHOICE } [\![\, E_1 \,]\!] \texttt{ [] } [\![\, E_2 \,]\!] \texttt{ [] ... [] } [\![\, E_n \,]\!] \texttt{ CHOICE> to } x$$

A simplification is possible if the type of x is `INT` and the choice is from a contiguous set of integers; then we have (where $m < n$)

$$[\![\, x := m, m+1, \ldots, n \,]\!] \;\rightsquigarrow\; \texttt{m n .. CHOICE to } x$$

7.5.2 Assignment to Array elements

Assignment to an array element requires some care, as the array element can be specified either by an integer denoting its position, as in `3 of ARRAY1`, or more often by some expression, probably involving a variable. The RVM word `to` is used for assignment, and will normally be followed by a variable name, to which the assignment is made. If an attempt is made to assign to a certain position in the array in this way, for instance

```
100 to INDEX of ARRAY1
```

this will result in the value 100 being assigned to `INDEX`. To avoid this, the words `<<` and `>>` are used around the index expression, so that whatever is in between them is interpreted as an array index — and not the target variable for `to`.

$$[\![\, a(i) := E \,]\!] \;\rightsquigarrow\; [\![\, E \,]\!] \texttt{ to << } [\![\, i \,]\!] \texttt{ >> of } a$$

If the array position *is* specified as an integer, the chevrons are not needed, so the translator, detecting this, could produce the simplified translation

$$[\![\, a(3) := E \,]\!] \;\rightsquigarrow\; [\![\, E \,]\!] \texttt{ to 3 of } a$$

7.6 Control Structures

7.6.1 Selection

We discussed the conventional IF-ELSE-THEN structures and their relationships to a definition in terms of choice and guard in section 2.1.5.

These correspond to the basic AMN forms and translations as follows.

$[\![\text{IF } g \text{ THEN } S \text{ END}]\!] \rightsquigarrow [\![g]\!]$ `IF` $[\![S]\!]$ `THEN`

Where an ELSE clause is included, the translation is

$[\![\text{IF } g \text{ THEN } S \text{ ELSE } T \text{ END}]\!] \rightsquigarrow [\![g]\!]$ `IF` $[\![S]\!]$ `ELSE` $[\![T]\!]$ `THEN`

In either case, the guards are complementary (including the implicit $\neg g$), so one choice will always be taken, and reversibility is not invoked (we examine the structures which incorporate infeasibility in the next section). As these constructs are in a reversible context, they may be encountered more than once in a backtracking program, perhaps with different guards being true.

These are defined in GSL in terms of guard and choice. The version with an ELSE clause is specified as

$$g \Longrightarrow S \,[]\, \neg g \Longrightarrow T$$

Where T is `skip` this corresponds to the shortform IF without an ELSE. These forms could in fact be translated directly using RVM's choice and guard features, such that

$$g \Longrightarrow S \,[]\, \neg g \Longrightarrow T$$

$\rightsquigarrow$

`<CHOICE` $[\![g]\!]$ `-->` $[\![S]\!]$ `[]` $[\![g]\!]$ `NOT -->` $[\![T]\!]$ `CHOICE>`

But this is a far less efficient way of implementation than to use the primitives `IF, ELSE` and `THEN` in RVM.

Generalising a simple two-way choice to those with three or more, there are different ways to express these. We first examine nested choices, which have a simple translation and have an order of evaluation for the guards built in. These rapidly become unwieldy, and it may be desirable to formulate the structure as a list of choices. In this case, since reversibility is not to be invoked, an "otherwise" choice must be included, albeit with merely a `skip` action. But the guards might still overlap — that is, more than one may be true in a given situation, and in a non-deterministic RB0 list such as example (7.5), on page 91, the order of evaluation is not specified. We introduce a provisional RB1 construct to specify an order-dependent list of guarded choices, which requires mutual exclusivity in the corresponding RB0, but maps onto an RVM `CASE` statement.

The basic IF selections seen above can be nested to any depth — that is, either S or T can be another IF statement, and so on within them (arranging the conditions so that the nesting is in the second one imparts an implicit order of evaluation of the guards).

For example,

$$\begin{aligned} & g_1 \Longrightarrow S_1 [] \\ & \neg g_1 \Longrightarrow \\ & \quad (g_2 \Longrightarrow S_2 [] \\ & \quad \neg g_2 \Longrightarrow \\ & \qquad (g_3 \Longrightarrow S_3 [] \\ & \qquad \neg g_3 \Longrightarrow S_4)) \end{aligned} \tag{7.4}$$

As all the nested conditions form complementary pairs, at least one true condition will be encountered. There is no requirement that only one be true; formally, it is sufficient that the whole statement (whichever action is invoked by a true guard) establishes the required postcondition with regard to the invariant.

Many programming languages provide an `ELSEIF` or `ELSIF` construct of some kind to make these more manageable and intelligible; B's AMN has an ELSIF structure. RVM Forth does not have this facility as such; we *could* translate directly as nested `IF`s, with the accompanying profusion of nesting as the number of choices grows. Alternatively, we propose a CASE construct for RB1 (an adaptation of the ELSIF, and unrelated to the existing AMN CASE) which would have an simple and intuitive translation to RVM, and a complementary translation to RB0.

So a possible translation of the semantic pattern above, using AMN, would be as below. In this case it can be seen that the RVM follows the GSL more closely, allowing for postfix and delimiting the `IF`s by `THEN`s.

$$\begin{array}{lcl} & & [\![g_1]\!] \ \texttt{IF} \ [\![S_1]\!] \\ & & \texttt{ELSE} \\ [\![\text{IF } g_1 \text{ THEN } S_1 & & \quad [\![g_2]\!] \ \texttt{IF} \ [\![S_2]\!] \\ \text{ELSIF } g_2 \text{ THEN } S_2 & & \quad \texttt{ELSE} \\ \text{ELSIF } g_3 \text{ THEN } S_3 & \leadsto & \qquad [\![g_3]\!] \ \texttt{IF} \ [\![S_3]\!] \\ \text{ELSE } S_4 & & \qquad \texttt{ELSE} \ [\![S_4]\!] \\ \text{END}]\!] & & \qquad \texttt{THEN} \\ & & \quad \texttt{THEN} \\ & & \texttt{THEN} \end{array}$$

The structure of example (7.4) gives an order to the evaluation of the guards, which is also the situation in the `CASE` statement of RVM. Were this possible with a simple list of choices of guarded commands in GSL, we would be able to map a generalised version of (7.4) to convert the nested `IF`s to a list of choices where the last is effectively an "otherwise" condition. This would *look* more like a case statement:

$$\begin{array}{l} g_1 \Longrightarrow S_1 [] \\ \neg g_1 \Longrightarrow \\ \quad (g_2 \Longrightarrow S_2 [] \\ \quad \neg g_2 \Longrightarrow \\ \quad \vdots \\ \qquad (g_{n-1} \Longrightarrow S_{n-1} [] \\ \qquad \neg g_{n-1} \Longrightarrow S_n) \ldots) \end{array} \quad \rightarrow \quad \begin{array}{l} g_1 \Longrightarrow S_1 [] \\ g_2 \Longrightarrow S_2 [] \\ \vdots \\ g_{n-1} \Longrightarrow S_{n-1} [] \\ \neg(g_1 \vee g_2 \vee \ldots \vee g_{n-1}) \Longrightarrow S_n \end{array} \tag{7.5}$$

However the ordinary GSL choice [] is non-deterministic, and cannot rely on order of evaluation in this way, so these two are *not* equivalent. To make the choices of the right-hand side commutative, we would have to ensure that exactly one guard is true; then the behaviour would correspond to that of the left-hand side. The guards must be made stronger, and the structure that would achieve this in RB0 is as follows. This is not something one would expect a human to write, but is quite appropriate for an automatic logic prover.

$$\begin{array}{l} g_1 \Longrightarrow S_1 [] \\ g_2 \wedge \neg g_1 \Longrightarrow S_2 [] \\ g_3 \wedge \neg(g_1 \vee g_2) \Longrightarrow S_3 [] \\ \vdots \\ g_{n-1} \wedge \neg(g_1 \vee g_2 \vee \ldots \vee g_{n-2}) \Longrightarrow S_{n-1} [] \\ \neg(g_1 \vee g_2 \vee \ldots \vee g_{n-1}) \Longrightarrow S_n \end{array} \tag{7.6}$$

We therefore propose a new CASE statement for RB1, based on the AMN[1] ELSIF, but having a direct translation to RVM. For proof analysis purposes, this would be converted to the RB0 version in example (7.6) above, which replaces the implicit ordering of choices with mutual exclusivity and exhaustiveness. Where the programmer was sure he was making a non-backtracking choice, he would use this construction.

$$\begin{array}{l} [\![\text{CASE} \\ \text{IF } g_1 \text{ THEN } S_1 \\ \text{IF } g_2 \text{ THEN } S_2 \\ \vdots \\ \text{IF } g_{n-1} \text{ THEN } S_{n-1} \\ \text{ELSE } S_n \\ \text{END}]\!] \end{array} \quad \rightsquigarrow \quad \begin{array}{l} \texttt{CASE} \\ [\![g_1]\!] \ \texttt{?OF} \ [\![S_1]\!] \ \texttt{ENDOF} \\ [\![g_2]\!] \ \texttt{?OF} \ [\![S_2]\!] \ \texttt{ENDOF} \\ \vdots \\ [\![g_{n-1}]\!] \ \texttt{?OF} \ [\![S_{n-1}]\!] \ \texttt{ENDOF} \\ [\![S_n]\!] \\ \texttt{ENDCASE} \end{array}$$

[1]There is an existing CASE in AMN, however it is merely a somewhat redundant re-expression of the SELECT statement, used in non-deterministic guarded choices.

7.6.2 Stand-alone Guards and Choice

The control structures in the previous section were non-reversing, as they would not encounter a situation in which none of the guards were true, rendering progress infeasible. But these are a subtype of more general guarded choice constructs where the event space need not be fully covered by the conditions, thus allowing infeasibility. Formally, the order in which the guards are evaluated is not specified, so they are also non-deterministic.

The most basic form can be thought of as having all its guards true. Then it is equivalent to a choice between a set of actions (bounded choice), represented in GSL for n substitutions S as

$$S_1 \, [] \, S_2 \, [] \ldots [] \, S_n$$

The AMN version translated to RVM is

$$[\![\text{CHOICE } S_1 \text{ OR } S_2 \text{ OR } \ldots \text{ OR } S_n \text{ END}]\!] \quad \rightsquigarrow$$

```
<CHOICE ⟦ S₁ ⟧ [] ⟦ S₂ ⟧ [] ... [] ⟦ Sₙ ⟧ CHOICE>
```

Here a subsequent infeasibility and reversal will cause another choice to be taken; if all choices lead to infeasibility, reversal continues back to any previous choices.

In section 5.1 it was mentioned that the RVM implementation for choice normally goes through the choices in the order as given. For a more rigorously non-deterministic bounded choice we can use the following translation, assuming the translator has counted the number of choices to n.

$$[\![S_1 \, [] \, S_2 \, [] \ldots [] \, S_n]\!] \quad \rightsquigarrow$$

```
1 n .. RANDOM-CHOICE CASE
1 OF ⟦ S₁ ⟧ ENDOF
2 OF ⟦ S₂ ⟧ ENDOF
⋮
n OF ⟦ Sₙ ⟧ ENDOF
ENDCASE
```

Should an infeasibility be encountered “downstream” of this, such that another choice of S would be desirable, the `RANDOM-CHOICE` will choose another number from 1 to n. Once these have run out, the reversal will continue back to the previous reversible choice, if applicable.

In the more general case for guarded choices, the GSL framework is

$$\begin{array}{l} g_1 \Longrightarrow S_1 [] \\ g_2 \Longrightarrow S_2 [] \\ \vdots \\ g_n \Longrightarrow S_n \end{array}$$

Any number including zero of the guards may be true, so the statement as a whole can be infeasible; the RVM therefore allows for this with a reversible translation. The AMN for this construct uses a SELECT statement.

```
SELECT $g_1$ THEN $S_1$
WHEN $g_2$ THEN $S_2$
⋮
WHEN $g_n$ THEN $S_n$
END
```

$\rightsquigarrow$

```
<CHOICE
  $[\![ g_1 ]\!]$ --> $[\![ S_1 ]\!]$ []
  $[\![ g_2 ]\!]$ --> $[\![ S_2 ]\!]$ []
⋮
  $[\![ g_n ]\!]$ --> $[\![ S_n ]\!]$
CHOICE>
```

7.6.3 Loops

In implementations, loops must contain a VARIANT and an INVARIANT section for proof purposes, to ensure termination and maintenance of the machine invariant. However, these play no part in the generated code, so the translation structure can be simplified.

$[\![$ WHILE g DO S END $]\!]$ $\rightsquigarrow$ `BEGIN` $[\![g]\!]$ `WHILE` $[\![S]\!]$ `REPEAT`

7.7 Sets, Ordered Pairs and Sequences

7.7.1 Enumerated Sets

These can occur as variables, or as sets in the SETS clause; in RVM they may be reversible, non-reversible or even constants. The basic translation is

$[\![\{e_1, e_2, \ldots, e_n\}]\!]$ $\rightsquigarrow$ *TYPE* `{` $[\![e_1]\!]$ `,` $[\![e_2]\!]$ `, ...` $[\![e_3]\!]$ `, }`

The comma in RVM is an operation to compile the element into the set — so each element including the final one requires a comma.

It may be desirable on occasion, for instance when discussing pair construction, to differentiate between the abstract “types” of B, and the corresponding “classes” of object constructed in the RVM. We have four basic classes of element which can be members of sets or of pairs in RVM, which are integer, string, pair, and set. These are all stored with a somewhat different internal structure.

For the construction of the set, its type must be specified, here denoted by the expression ***TYPE***. Unlike the maximal sets which a type denotes in B, a type in the RVM is an empty set, and the RVM code uses the words `INT`, `STRING`, `PROD` and `POW` to build up an empty set of the required type on the stack. The word $\boxed{\texttt{\{}}$, which begins construction of a set, takes this value from the stack. This mechanism can build sets of arbitrary complexity for elements of the required class.

The question of when in the development intended strings become actual strings is not entirely straightforward; strings have been provided by a limited Library machine in B. We assume for the purposes of this document that a set from the SETS clause of the specification machine would pass through the translation process in this way:

$RESPONSE = \{yes,\ no,\ maybe\} \ \rightsquigarrow$

```
STRING { " yes" , " no" , " maybe" , } VALUE RESPONSE
```

7.7.2 Choice from a Set

For the most part, a choice from a set is required to be reversible. There are two possible translations, one using the simpler `CHOICE` operation, which takes choices in the same order as the set is internally represented; and `RANDOM-CHOICE`, which randomises the order.

$[\![x :\in S]\!] \ \rightsquigarrow\ [\![S]\!]$ `CHOICE to` x

7.7.3 Set Comprehension

While a number of set comprehensions are possible in B, many do not concern us as they will result in infinite sets which we would not want to construct. A more common sort, however, is a finite set which might be required during an operation, or to be its output. This will probably be specified as

$\{x \mid x \in S \wedge P\}$

where S is an already defined set, and P is a constraining predicate on x.

In our expanded RB0, the Prospective Values syntax can be used to specify this more procedurally in an implementation, and translated into RVM.

For semantic reference to types, we use the notation $[\![v]\!]^T$ or $[\![S]\!]^T$ to stand for the type of a variable v or for the type of an *element* of the set S in the translation (which will use the words `INT, STRING, POW` and `PROD`).

$[\![\{x :\in S;\ P \Longrightarrow \mathsf{skip} \diamond x\}]\!] \ \rightsquigarrow$

$[\![S]\!]^T$ `{ <RUN`
  $[\![S]\!]$ `CHOICE to` x
  $[\![P]\!]$ `-->` x `RUN> }`

7.7.4 Ordered Pairs

At specification level, ordered pairs can be represented as (x, y), or sometimes just x, y. However at the translation level, a less ambiguous notation would use the maplet symbol $x \mapsto y$.

In the context of a set construction as above, the translation is relatively simple as the type information for the enclosing set will allow the RVM to infer the classes required for construction using only the maplet creator `|->` . Supposing the specification was for a set of pairs from sets S and T, with each element separately satisfying some predicate P. One way of specifying this is

$$\{x, y \mid x \in S \wedge y \in T \wedge P\}$$

The corresponding RB0, using the diamond constructor, and its translation would be

$$[\![\, \{x :\in S;\;\; y :\in T;\;\; P \Longrightarrow \mathsf{skip} \diamond x \mapsto y\} \,]\!]$$

$\rightsquigarrow$

$[\![S]\!]^T$ $[\![T]\!]^T$ `PROD { <RUN`
  $[\![S]\!]$ `CHOICE to` x
  $[\![T]\!]$ `CHOICE to` y
  $[\![P]\!]$ `-->` $x\ y$ `|-> RUN> }`

Type inference rules, as in Abrial [1], can be used to infer the types of x and y, and via the set axiom

$$x \in S \wedge y \in T \Leftrightarrow x \mapsto y \in S \times T$$

the type of the resulting set. This is translated into the class information

$[\![S]\!]^T$ $[\![T]\!]^T$ `PROD`

which allows the maplet creator `|->` to construct a maplet of the correct class for each element of the final set.

Constructing pairs *without* type information provided in this way requires that the RVM know the class of each part in order to use the correct maplet constructor. There are four classes: integers, strings, sets and pairs. These lead to sixteen possible constructors; using the symbolic suffixes `I, $, S` and `P` to indicate integer, string, set and pair respectively, these constructors are:

```
|->I,I  |->I,$  |->I,S  |->I,P
|->$,I  |->$,$  |->$,S  |->$,P
|->S,I  |->S,$  |->S,S  |->S,P
|->P,I  |->P,$  |->P,S  |->P,P
```

Using the notation $[\![x]\!]^S$ to stand for the class *symbol* in this context, such that $[\![x]\!]^S \in$ `{I,$,S,P}`, where these specifications are required the translation is

$x \mapsto y \;\; \rightsquigarrow \;\; x\ y$ `|->`$[\![x]\!]^S$`,`$[\![y]\!]^S$

7.7.5 Sequences

Sequences are only likely to be explicitly constructed, and their translation is very similar to that of enumerated sets, but using the operations `[ ]` as delimiters rather than `{ }` . Again, the comma is used to compile the elements into the sequence, and the expression *TYPE* will be a postfix expression for the type of each element.

$$[\![\, [e_1, e_2, \ldots, e_n] \,]\!] \quad \rightsquigarrow \quad \mathit{TYPE}\ \texttt{[}\ [\![\, e_1 \,]\!]\ \texttt{,}\ [\![\, e_2 \,]\!]\ \texttt{,}\ \ldots\ [\![\, e_n \,]\!]\ \texttt{,}\ \texttt{]}$$

Some sequence operations which are in prefix form with bracketed arguments become postfix operations in RVM. For a sequence S:

$$[\![\, \mathsf{head}(S) \,]\!] \quad \rightsquigarrow \quad [\![\, S \,]\!]\ \texttt{HEAD}$$
$$[\![\, \mathsf{tail}(S) \,]\!] \quad \rightsquigarrow \quad [\![\, S \,]\!]\ \texttt{TAIL}$$
$$[\![\, \mathsf{front}(S) \,]\!] \quad \rightsquigarrow \quad [\![\, S \,]\!]\ \texttt{FRONT}$$
$$[\![\, \mathsf{last}(S) \,]\!] \quad \rightsquigarrow \quad [\![\, S \,]\!]\ \texttt{LAST}$$

Concatenating two sequences is:

$$[\![\, S \frown T \,]\!] \quad \rightsquigarrow \quad [\![\, S \,]\!]\ [\![\, T \,]\!]\ \texttt{^}$$

Appending and prepending a single element e are respectively translated as follows:

$$[\![\, S \leftarrow e \,]\!] \quad \rightsquigarrow \quad [\![\, S \,]\!]\ [\![\, e \,]\!]\ \texttt{<-}$$
$$[\![\, e \rightarrow S \,]\!] \quad \rightsquigarrow \quad [\![\, e \,]\!]^T\ \texttt{[}\ [\![\, e \,]\!]\ \texttt{,}\ \texttt{]}\ [\![\, S \,]\!]\ \texttt{^}$$

In the latter case, there is no built-in primitive for prepending, so a sequence is constructed containing the single element, and concatenated with the existing sequence.

7.7.6 Relational Image and Function Application

In the image, R is the relation, while S is the set generating the image.

$$[\![\, R[S] \,]\!] \quad \rightsquigarrow \quad [\![\, R \,]\!]\ [\![\, S \,]\!]\ \texttt{IMAGE}$$
$$[\![\, F(E_0) \,]\!] \quad \rightsquigarrow \quad [\![\, F \,]\!]\ [\![\, E_0 \,]\!]\ \texttt{APPLY}$$

This is function application as a special case of a relational image, and would be used where the function was specified as an enumerated set of maplets, rather than an operation taking its arguments from the stack; so here E_0 is a single expression of the same type as the function domain.

7.8 Prospective Value Computations

The theory of PV semantics (section 4.4) and their implementation in RVM (section 5.1.1) have been considered; we also looked at their use in set comprehension expressions above, section 7.7.3.

The general usage for deterministic operations, that is where $S \diamond E$ returns only a single value which need not be enclosed in a set, depends on the type of E. In these cases the `<RUN` delimiter is paired with a keyword containing the type of E, so there are four possibilities.

- Where E is an integer expression,

  $[\![S \diamond E]\!]$ $\rightsquigarrow$ `<RUN` $[\![S]\!]$ $[\![E]\!]$ `INT>`

- Where E is a string,

  $[\![S \diamond E]\!]$ $\rightsquigarrow$ `<RUN` $[\![S]\!]$ $[\![E]\!]$ `STRING>`

- Where E is a set

  $[\![S \diamond E]\!]$ $\rightsquigarrow$ `<RUN` $[\![S]\!]$ $[\![E]\!]$ `SET>`

- Where E is a pair,

  $[\![S \diamond E]\!]$ $\rightsquigarrow$ `<RUN` $[\![S]\!]$ $[\![E]\!]$ `PAIR>`

A further use is in a situation where a number of values may be returned satisfying some particular condition — for instance several solutions to a problem, but only one of these values (any one) is actually required. In such cases, the first value found would be adequate.

Rather than use a full $S \diamond E$ construction to collect all the solutions in a set, we can use the specification $S \diamond_1 E$. This will return one of the possible solutions, not in a set, without affecting the state (as is usual for PV constructs), and also performing garbage collection on the search.

The translation uses the `<COLLECT ... TILL ... SATISFIED>` construction in RVM, as shown in section 5.1, in a slightly different way:

$[\![S \diamond_1 E]\!]$ $\rightsquigarrow$
$[\![E]\!]^T$ `{ <COLLECT` $[\![S]\!]$ $[\![E]\!]$ `TILL CARD SATISFIED> } CHOICE`

The word `CARD` here tests the cardinality of the set which is left on the stack by `TILL`. Since the set will be non-empty after the first result is found, the value of 1 is taken as "true" by `SATISFIED>` which then abandons any further search. Then `CHOICE` picks out this result and leaves it on the stack.

In section 7.5.1 we considered a form of non-deterministic assignment wherein the right-hand side is a bunch of values. Since the unadorned $S \diamond E$ results in a bunch, we can also admit assignment statements of the form

$$x := S \diamond E$$

where x is assigned one of the values in the bunch resulting from the PV computation. However, the latter can result in the null bunch, which one would expect to signal an infeasible assignment, and to trigger reversal; the RVM does this, as the translation becomes assignment from a set, and choice from an empty set triggers reversal.

$[\![x := S \diamond E]\!]$ $\rightsquigarrow$ $[\![E]\!]^T$ `{` $[\![S \diamond E]\!]$ `} CHOICE to` x

7.9 Operation Definition and Invocation

7.9.1 Definition

There are two possible ways of translating these, as we discussed in section 5.3. One is to follow a C-like model, using pass-by-reference to assign values to the output parameters. However, this arises because of a limitation in C which RVM does not have, and can be replaced by a model where the input values are taken from the stack, and the output values left on the stack.

In the example below, the operation *Opname* has n inputs x and m outputs y; when this operation is called by another operation, it is required that the output values be in the expected order for assignment to the variables of the calling operation. We use a list r for the latter group of variables. The definition translation is

$$[\![y_1, y_2, \ldots, y_m \longleftarrow Opname(x_1, x_2, \ldots, x_n) \ \hat{=}\ S]\!]$$

$\rightsquigarrow$

$$\begin{array}{l} \texttt{:}\ Opname\ \texttt{(: VALUE\_}\ \ x_1\ \texttt{VALUE\_}\ \ x_2 \ldots \texttt{VALUE\_}\ \ x_n\ \texttt{:)} \\ \quad 0\ \texttt{VALUE\_}\ \ y_1\ 0\ \texttt{VALUE\_}\ \ y_2 \ldots 0\ \texttt{VALUE\_}\ \ y_m \\ \quad [\![S]\!]\ y_1\ y_2 \ldots y_m\ \texttt{;} \end{array}$$

There are some simplifications which can be made, as either the input or output list might be empty. If local variables are still required in the body of the operation S, then an empty parameter list can be used:

$$[\![Opname \ \hat{=}\ S]\!] \rightsquigarrow \texttt{:}\ Opname\ \texttt{(: :)}\ [\![S]\!]\ \texttt{;}$$

If S does not contain any local variable declarations, there is no requirement to construct and dismantle a stack frame, and the following form is more efficient.

$$[\![Opname \ \hat{=}\ S]\!] \rightsquigarrow \texttt{:}\ Opname\ [\![S]\!]\ \texttt{;}$$

7.9.2 Invocation

The invocation may or may not include assignment. The list of variables r here belongs to the calling environment, and has m members, the same as the y list of outputs left on the stack in the above operation.

$$[\![r_1, r_2, \ldots, r_m \longleftarrow Opname(x_1, x_2, \ldots, x_n)]\!]$$

$\rightsquigarrow$

$$x_1\ x_2 \ldots x_n\ Opname\ \texttt{to}\ r_m\ \texttt{to}\ r_{m-1} \ldots \texttt{to}\ r_1$$

7.10 Local Variables

There are three forms for local variables in GSL, with slightly different AMN forms. The effective difference is that one has the variables initialised to some value; since this is done by default in RVM (if only to a type-agnostic `NULL` value), we can treat these as one. The GSL and AMN forms are

$@x.S$	VAR x IN S END
$@x.(P \Longrightarrow S)$	ANY x WHERE P THEN S END
$@x.(x = E \Longrightarrow S)$	LET x BE $x = E$ IN S END

In the first, it is assumed that x is assigned a value in S consistent with its usage. In the second, a type is specified, but not necessarily a value, and in the third the variable is actually initialised to a value. The value will probably change during S, of course.

In the RVM translation, then, all are equivalent to the third version. We have only to consider the scope of the variables, which is S. If S is the whole operation, the scope declaration is unnecessary, and its removal is another possible optimisation. The scalar translation is

$[\![\text{VAR } x \text{ IN } S \text{ END }]\!] \rightsquigarrow$

`(SCOPE 0 VALUE_` $x\ [\![S]\!]$ `SCOPE)`

The single variable here can be expanded to a list in the same way that the list in the operation definition (section 7.9.1 above) is so expanded.

A 2008 paper presented at the Euroforth 2008 conference describes a method of compiling from Forth (specifically the RVM), using a two-pass compiling technique with an intermediate language which carries type information. This technique has fed back into work which will further the work of this thesis. The paper is included as Appendix G.

7.11 Summary

In this chapter we have dealt with Objective 7, the provision of a set of translation schemas for RB0 operations to RVM, noting that, as mentioned in chapter 4, the RB0 is at this stage not in a "pure" GSL-style notation, but has some AMN constructs for clarity. These will ultimately be incorporated into RB1. The RB0 and RVM used is summarised in Appendix B.

In the next, penultimate chapter, we look at the issue of relating the foundations of Bunch theory to those of Set theory, to address the perceived lack of a formal model by some critics.

Chapter 8

Theoretical Underpinnings for Bunch Theory

The thread of Bunch Theory runs through the previous chapters in various ways. As its roots are in Set theory, the latter having a well-established structure of axioms and theorems, we thought it would aid the rigour of proofs if this relationship were grounded in a Denotational Semantics which explicitly linked Bunch and Set theories. Bunch theory departs from Set theory in treating the concepts of collection and packaging as independent and separable; sometimes a negative attitude to this is discernible in reviews, for instance.

Using the formal relationship defined in the denotational semantics, bunch postulates could be disassembled and translated into the corresponding set-theoretic expressions, which must then correspond to theorems in Set theory. For instance, an equivalence postulate in bunch form can have its left-hand side translated, and this set-theoretic form would then translate back — via the appropriate theorem — into the right-hand side of the bunch equivalence. As a shorthand, we may refer to the bunch "world" in contrast to the semantic "world". Some examples are worked through in a later section.

8.1 Denotational Relation to Set Theory

We begin with a reminder of the basic notation and relationships from Chapter 2. In the following table, table 8.1, $\boldsymbol{A}$ and $\boldsymbol{B}$ are bunches, and the empty bunch is **null**. Since the comma is now an operation representing bunch union, we use the maplet symbol $\mapsto$ to create ordered pairs, and we generally reserve brackets for expressing the precedence of operations.

As regards precedence to reduce the use of brackets, the convention has the maplet as the highest, followed by bunch intersection, union, subtraction, and bunch inclusion.

To construct the basic denotational relationships, two kinds of bunch expressions are defined: bunch value expressions and bunch predicates, which may be true or false. Then a set-theoretic meta-language denotes the meaning of these within superscripted semantic brackets as follows.

Bunch Expression	Definition
A, B	Union of A and B. Commutative.
$A, \mathbf{null} = A$	**null** is identity for union
$A \setminus B$	Elements of A not in B
$A{'}B$	Intersection of A and B
$A{'}\mathbf{null} = \mathbf{null}$	Absorption by **null** under intersection.
$A : B$	A is a sub-bunch of B
$¢A$	Cardinality of A

Table 8.1: Basic bunch Notations

Where e is any bunch value expression, we write $[\![e]\!]^{\nu}$ for its semantic value denotation in the meta-language, and where p is a bunch predicate, we write $[\![p]\!]^{\tau}$. An elemental bunch a is represented semantically as the set containing it, $\{a\}$. These and some basic relationships are set out in table 8.2.

Bunch Expression	Semantic Definition
e	$[\![e]\!]^{\nu}$
p	$[\![p]\!]^{\tau}$
$\mathbf{null}$	$\varnothing$
a	$\{a\}$
e_1, e_2	$[\![e_1]\!]^{\nu} \cup [\![e_2]\!]^{\nu}$
$e_1{'}e_2$	$[\![e_1]\!]^{\nu} \cap [\![e_2]\!]^{\nu}$
$e_1 \mapsto e_2$	$[\![e_1]\!]^{\nu} \times [\![e_2]\!]^{\nu}$
$\{e\}$	$\{[\![e]\!]^{\nu}\}$

Table 8.2: Basic bunch denotations

The bunch union and intersection translate in a natural way to set notation; we can derive the semantic expression via lower level transformations. With a and b as elemental bunches, and $\leadsto$ denoting such a transformation (from one denotational representation to another), while $\Rightarrow, \Leftrightarrow$ or $=$ implies a standard set-theoretic or logic result, we can follow through a complete translation using the rules in table 8.2 (and later, tables 8.3 and 8.4). For instance:

$$\begin{aligned} [\![a, b]\!]^{\nu} &\leadsto [\![a]\!]^{\nu} \cup [\![b]\!]^{\nu} \\ &\leadsto \{a\} \cup \{b\} \\ &= \{a, b\} \end{aligned}$$

The set at the end is naturally the transformation of the initial bunch a, b.

The set theory results used in the following examples are listed below.

$$\{e_0\} \cap (\{e_1\} \cup \{e_2\}) = \{e_0\} \cap \{e_1\} \cup \{e_0\} \cap \{e_2\} \tag{8.1}$$
$$\{e_1\} \cup \{e_2\} \subseteq \{e_3\} \Leftrightarrow \{e_1\} \subseteq \{e_3\} \wedge \{e_2\} \subseteq \{e_3\} \tag{8.2}$$
$$\{e_1\} \subseteq \{e_2\} \cap \{e_3\} \Leftrightarrow \{e_1\} \subseteq \{e_2\} \wedge \{e_1\} \subseteq \{e_3\} \tag{8.3}$$
$$\mathrm{card}\,(\{e_1\} \cup \{e_2\}) + \mathrm{card}\,(\{e_1\} \cap \{e_2\}) = \mathrm{card}\,(\{e_1\}) + \mathrm{card}\,(\{e_2\}) \tag{8.4}$$
$$\{e_1\} \subseteq \{e_2\} \Rightarrow \mathrm{card}(\{e_1\}) \leq \mathrm{card}(\{e_2\}) \tag{8.5}$$
$$\neg\{a\} \subseteq \{e\} \Rightarrow \mathrm{card}(\{e\} \cap \{a\}) = \mathrm{card}(\varnothing) = 0 \tag{8.6}$$

The maplet construction $a \mapsto (b, c)$, where again a, b and c are elemental, is transformed thus (using the above bunch result $a, b \rightsquigarrow \{a, b\}$.):

$$\begin{aligned} [\![a \mapsto (b,c)]\!]^{\nu} &\rightsquigarrow [\![a]\!]^{\nu} \times [\![b,c]\!]^{\nu} \\ &\rightsquigarrow \{a\} \times \{b,c\} \\ &= \{a \mapsto b, a \mapsto c\} \end{aligned}$$

Deriving the same set expression from another bunch expression would show that the bunch expressions are equivalent. Here we use

$$\begin{aligned} [\![a \mapsto b, a \mapsto c]\!]^{\nu} &\rightsquigarrow [\![a \mapsto b]\!]^{\nu} \cup [\![a \mapsto c]\!]^{\nu} \\ &\rightsquigarrow ([\![a]\!]^{\nu} \times [\![b]\!]^{\nu}) \cup ([\![a]\!]^{\nu} \times [\![c]\!]^{\nu}) \\ &\rightsquigarrow (\{a\} \times \{b\}) \cup (\{a\} \times \{c\}) \\ &= \{a \mapsto b\} \cup \{a \mapsto c\} \\ &= \{a \mapsto b, a \mapsto c\} \end{aligned}$$

We find that we have a distributive law for a maplet between a bunch and an element, that is

$$a \mapsto (b, c) = a \mapsto b, a \mapsto c$$

This extends in a similar way to the construction of a maplet between two non-elemental bunches, which gives, for instance:

$$(1, 2) \mapsto (3, 4) = 1 \mapsto 3, 1 \mapsto 4, 2 \mapsto 3, 2 \mapsto 4$$

The derivation would proceed via $\{1, 2\} \times \{3, 4\}$, instantiating from the right-hand side of table 8.2.

Suppose we wish to show a distributive rule for intersection over union,

$$e_0\,'(e_1, e_2) = e_0\,'e_1, e_0\,'e_2$$

This would proceed as follows, using the established set theory law (referenced to the list on page 102). Note, in set and bunch notation, unions have lower precedence than intersection.

$$\begin{aligned} [\![e_0\,'(e_1, e_2)]\!]^{\nu} &\rightsquigarrow [\![e_0]\!]^{\nu} \cap [\![e_1, e_2]\!]^{\nu} \\ &\rightsquigarrow [\![e_0]\!]^{\nu} \cap ([\![e_1]\!]^{\nu} \cup [\![e_2]\!]^{\nu}) \\ &\rightsquigarrow \{e_0\} \cap (\{e_1\} \cup \{e_2\}) \\ &= \{e_0\} \cap \{e_1\} \cup \{e_0\} \cap \{e_2\} \quad \text{by Result (8.1)} \end{aligned}$$

In the other direction, we begin with $e_0\,'e_1, e_0\,'e_2$ and find:

$$\begin{aligned}
[\![e_0\,'e_1, e_0\,'e_2]\!]^\nu &\leadsto [\![e_0\,'e_1]\!]^\nu \cup [\![e_0\,'e_2]\!]^\nu \\
&\leadsto [\![e_0]\!]^\nu \cap [\![e_1]\!]^\nu \cup [\![e_0]\!]^\nu \cap [\![e_2]\!]^\nu \\
&\leadsto \{e_0\} \cap \{e_1\} \cup \{e_0\} \cap \{e_2\}
\end{aligned}$$

Some of the other denotations require a little more explanation. The denotation of $\{e\}$ is $\{[\![e]\!]^\nu\}$, but e might already be a set — for instance $\{1,2\}$. In this case, $\{[\![e]\!]^\nu\}$ becomes $\{\{1,2\}\}$.

The *contents* of a bunch which is a set (written $\sim\!e$) would intuitively be the bunch or bunch expression in the set. We can specify this in two ways. Given the bunch above, where $e = \{1,2\}$, the operation

$$\mathsf{choice}\,([\![e]\!]^\nu) \quad \text{gives} \quad \mathsf{choice}\{\{1,2\}\} = \{1,2\}$$

Evaluated thus in the set environment, this becomes the bunch 1,2 in the bunch "world". Another way is to use the *definite description* or unique value quantifier, written ι or sometimes μ (we use the former). This yields

$$\iota x \bullet x \in [\![e]\!]^\nu$$

That is, the unique value which is an element of $\{\{1,2\}\}$.

And the "contents" of a bunch expression which is *not* a set is defined as the **null** bunch.

These and some other common constructs are summarised in table 8.3.

Bunch Expression	Semantic Definition	Remarks
$¢(e)$	$\mathrm{card}\,([\![e]\!]^\nu)$	Cardinality
$\sim\!e$	$\iota x \bullet x \in [\![e]\!]^\nu$	e is a set; ι unique value
$\sim\!e$	$\mathsf{choice}\,([\![e]\!]^\nu)$	where $\mathsf{choice}(S) \in S$
$g \longrightarrow e$	$\{y \mid [\![g]\!]^\tau \wedge y \in [\![e]\!]^\nu\}$	g is a guard (predicate)
$\S x \bullet g \longrightarrow e$	$\{x, y \mid [\![g]\!]^\tau \wedge y \in [\![e]\!]^\nu \bullet y\}$	
$e_1 = e_2$	$[\![e_1]\!]^\nu = [\![e_2]\!]^\nu$	
$e_1 : e_2$	$[\![e_1]\!]^\nu \subseteq [\![e_2]\!]^\nu$	Bunch Inclusion
$e_1 \in e_2$	$[\![e_1]\!]^\nu : [\![\sim\!e_2]\!]^\nu$	e_1 right type for set e_2

Table 8.3: Further bunch denotations

The bunch guard becomes a restriction within a set comprehension, with the guard g represented by its predicate denotation $[\![g]\!]^\tau$. The bunch comprehension denotation is a set comprehension of values taken by y in e as x ranges over values which satisfy g.

In the final line we have e_2 as a set, and e_1 as some element of an appropriate type for the set. This is defined in terms of bunch inclusion *after* applying the contents operator to e_2. The expression in the middle column could be further analysed at a lower level as

$$[\![e_1]\!]^\nu \subseteq \iota x \bullet x \in [\![e_2]\!]^\nu$$

In the last three lines of table 8.3, the entries in the left-hand column are actually bunch predicates; we add a table 8.4 which makes this explicit and also shows that the common connectives have the denotation one might expect.

Bunch Predicate Expression	Semantic Decomposition
$[\![e_1 = e_2]\!]^\tau$	$[\![e_1]\!]^\nu = [\![e_2]\!]^\nu$
$[\![e_1 : e_2]\!]^\tau$	$[\![e_1]\!]^\nu \subseteq [\![e_2]\!]^\nu$
$[\![e_1 \in e_2]\!]^\tau$	$[\![e_1]\!]^\nu : \ [\![\sim e_2]\!]^\nu$
$[\![e_1 \wedge e_2]\!]^\tau$	$[\![e_1]\!]^\tau \wedge [\![e_2]\!]^\tau$
$[\![e_1 \vee e_2]\!]^\tau$	$[\![e_1]\!]^\tau \vee [\![e_2]\!]^\tau$
$[\![\neg e]\!]^\tau$	$\neg[\![e]\!]^\tau$

Table 8.4: Bunch predicate denotations

We can derive a number of further bunch axioms via these semantic transformations and set theory. For instance, distributivity of inclusion:

$$\begin{aligned}
[\![e_1, e_2 : e_3]\!]^\tau &\rightsquigarrow [\![e_1, e_2]\!]^\nu \subseteq [\![e_3]\!]^\nu \\
&\rightsquigarrow [\![e_1]\!]^\nu \cup [\![e_2]\!]^\nu \subseteq [\![e_3]\!]^\nu \\
&\rightsquigarrow \{e_1\} \cup \{e_2\} \subseteq \{e_3\} \\
&\Leftrightarrow \{e_1\} \subseteq \{e_3\} \wedge \{e_2\} \subseteq \{e_3\} \quad \text{by Result (8.2)}
\end{aligned}$$

and then

$$\begin{aligned}
[\![e_1 : e_3 \wedge e_2 : e_3]\!]^\tau &\rightsquigarrow [\![e_1 : e_3]\!]^\nu \wedge [\![e_2 : e_3]\!]^\nu \\
&\rightsquigarrow [\![e_1]\!]^\nu \subseteq [\![e_3]\!]^\nu \wedge [\![e_2]\!]^\nu \subseteq [\![e_3]\!]^\nu \\
&\rightsquigarrow \{e_1\} \subseteq \{e_3\} \wedge \{e_2\} \subseteq \{e_3\}
\end{aligned}$$

Thus we have

$$e_1, e_2 : e_3 \Leftrightarrow e_1 : e_3 \wedge e_2 : e_3$$

as a basic bunch property. Similarly, we can derive

$$e_1 : e_2{}'e_3 \Leftrightarrow e_1 : e_2 \wedge e_1 : e_3$$

$$\begin{aligned}
[\![e_1 : e_2{}'e_3]\!]^\tau &\rightsquigarrow [\![e_1]\!]^\nu \subseteq [\![e_2{}'e_3]\!]^\nu \\
&\rightsquigarrow [\![e_1]\!]^\nu \subseteq [\![e_2]\!]^\nu \cap [\![e_3]\!]^\nu \\
&\rightsquigarrow \{e_1\} \subseteq \{e_2\} \cap \{e_3\} \\
&\Leftrightarrow \{e_1\} \subseteq \{e_2\} \wedge \{e_1\} \subseteq \{e_3\} \quad \text{by Result (8.3)}
\end{aligned}$$

And in the other direction

$$\begin{aligned}
[\![e_1 : e_2 \wedge e_1 : e_3]\!]^{\tau} &\rightsquigarrow [\![e_1 : e_2]\!]^{\nu} \wedge [\![e_1 : e_3]\!]^{\nu} \\
&\rightsquigarrow [\![e_1]\!]^{\nu} \subseteq [\![e_2]\!]^{\nu} \wedge [\![e_1]\!]^{\nu} \subseteq [\![e_3]\!]^{\nu} \\
&\rightsquigarrow \{e_1\} \subseteq \{e_2\} \wedge \{e_1\} \subseteq \{e_3\}
\end{aligned}$$

Two examples dealing with cardinality: one shows that

$$¢(e_1, e_2) + ¢(e_1 \,'\, e_2) = ¢\, e_1 + ¢\, e_2$$

Arithmetic operators are assumed to have their usual meanings when applied, as here, to elementary (value) bunches. This also goes for inequalities such as $\leq$, as we only apply them to cardinalities in this context. First we have

$$\begin{aligned}
[\![¢(e_1, e_2) + ¢(e_1 \,'\, e_2)]\!]^{\nu} &\rightsquigarrow \mathrm{card}\,([\![e_1, e_2]\!]^{\nu}) + \mathrm{card}\,([\![e_1 \,'\, e_2]\!]^{\nu}) \\
&\rightsquigarrow \mathrm{card}\,([\![e_1]\!]^{\nu} \cup [\![e_2]\!]^{\nu}) + \mathrm{card}\,([\![e_1]\!]^{\nu} \cap [\![e_2]\!]^{\nu}) \\
&\rightsquigarrow \mathrm{card}\,(\{e_1\} \cup \{e_2\}) + \mathrm{card}\,(\{e_1\} \cap \{e_2\}) \\
&= \mathrm{card}\,(\{e_1\}) + \mathrm{card}\,(\{e_2\}) \qquad \text{by Result (8.4)}
\end{aligned}$$

and in the other direction,

$$\begin{aligned}
[\![¢\, e_1 + ¢\, e_2]\!]^{\nu} &\rightsquigarrow [\![¢\, e_1]\!]^{\nu} + [\![¢\, e_2]\!]^{\nu} \\
&\rightsquigarrow \mathrm{card}\,([\![e_1]\!]^{\nu}) + \mathrm{card}\,([\![¢\, e_2]\!]^{\nu}) \\
&\rightsquigarrow \mathrm{card}\,(\{e_1\}) + \mathrm{card}\,(\{e_2\})
\end{aligned}$$

and one relating inclusion to cardinality, which shows that

$$e_1 : e_2 \Rightarrow ¢\, e_1 \leq ¢\, e_2$$

From $e_1 : e_2$ we obtain

$$\begin{aligned}
[\![e_1 : e_2]\!]^{\tau} &\rightsquigarrow [\![e_1]\!]^{\nu} \subseteq [\![e_2]\!]^{\nu} \\
&\rightsquigarrow \{e_1\} \subseteq \{e_2\} \\
&\Rightarrow \mathrm{card}(\{e_1\}) \leq \mathrm{card}(\{e_2\}) \qquad \text{by Result (8.5)}
\end{aligned}$$

and from the other direction,

$$\begin{aligned}
[\![¢\, e_1 \leq ¢\, e_2]\!]^{\tau} &\rightsquigarrow [\![¢\, e_1]\!]^{\nu} \leq [\![¢\, e_2]\!]^{\nu} \\
&\rightsquigarrow \mathrm{card}([\![e_1]\!]^{\nu}) \leq \mathrm{card}([\![e_2]\!]^{\nu}) \\
&\rightsquigarrow \mathrm{card}(\{e_1\}) \leq \mathrm{card}(\{e_2\})
\end{aligned}$$

Finally, in the following, a is an elementary bunch, so $\{a\}$ is a singleton set. This will show

$$\neg(a : e) \Rightarrow ¢(e \,'\, a) = 0$$

It also uses the fact that $\mathrm{card}(\varnothing) = 0$. Firstly,

$$\begin{aligned}
[\![\neg(a : e)]\!]^{\tau} &\rightsquigarrow \neg[\![a : e]\!]^{\tau} \\
&\rightsquigarrow \neg[\![a]\!]^{\nu} \subseteq [\![e]\!]^{\nu} \\
&\rightsquigarrow \neg\{a\} \subseteq \{e\} \\
&\Rightarrow \mathrm{card}(\{e\} \cap \{a\}) = \mathrm{card}(\varnothing) = 0 \qquad \text{by Result (8.6)}
\end{aligned}$$

and then

$$\begin{aligned}
[\![\cent(e\,'a) = 0]\!]^{\tau} &\rightsquigarrow [\![\cent(e\,'a)]\!]^{\nu} = 0 \\
&\rightsquigarrow \mathrm{card}\left([\![e\,'a]\!]^{\nu}\right) = 0 \\
&\rightsquigarrow \mathrm{card}\left([\![e]\!]^{\nu} \cup \{a\}\right) = 0 \\
&\rightsquigarrow \mathrm{card}(\{e\} \cap \{a\}) = 0
\end{aligned}$$

8.2 Summary

In this chapter we have addressed Objective 8, by beginning a process of formalising Bunch theory, utilising a Denotational approach to relate its theoretical underpinnings to those of its "parent" theory, Set theory.

In the final chapter we present our conclusions, a summary of the objectives reached, and a section on future work in the area.

Chapter 9

Conclusions and Further Work

In this thesis we have investigated how B's formal development language at the implementation level (B0) can be adapted to take advantage of a facility for logically reversible computations, the latter implemented in a virtual machine with facilities to execute reversible programs (the RVM). We have investigated the use of programming structures suggested by reversibility, and the relaxation of some constraints imposed on B0 by the need for it to be confined to feasible programs; the original theory allows infeasibility in the initial proof analysis stage, but not at the executable stage. These measures can result in a more expressive implementation language, RB0, which remains logically provable owing to the underlying mathematical properties of the new constructs, which have been formalised in [32, 33, 31].

We began, after introducing the preliminary relevant concepts of B, GSL, Bunch Theory, and Forth, with a chapter reviewing the early work on the physical basis for reversible computing. This had its roots in thermodynamic concerns about power dissipation and the ultimate lower limit on power consumption in computing. In the 1960s this limit was far less than the actual hardware was capable of, but in the 21st century, we find it coming ever closer — as long as non-reversible architectures are used. We saw that the total required dissipation, depending on the original figure of $kT \ln 2$ per erased bit of information, can be reduced by introducing reversible computations, as the dissipation is linked to the loss of information in irreversible computations. Reducing the number of bits irreversibly erased then reduces the total required energy dissipation. The physical reversibility so required implies a need for logical reversibility in new programming paradigms. Some irreversible steps are likely to be required still, for instance in the initialisation of a stack; but we demonstrated that an assignment statement (the basis for memory updates) can be made stepwise reversible, and thus in itself would at least theoretically require no extra expenditure of energy.

We then examined the changes required to allow hitherto forbidden commands and constructs to be used in RB0, along with the new constructs inspired by reversible computing. Use of the latter involves revocation of Dijkstra's "Law of the Excluded Miracle" [6], which disallowed infeasible operations in his Guarded Command Language GCL, and his weakest precondition calculus. Greg Nelson [21], while lamenting the loss of one of the best names ever given to a "law" of computing, nevertheless saw that the rigid imposition of this law interfered with the analysis of programs (where the ability to contravene the law of the excluded miracle enables us to decompose a conditional into choice and guard constructs) and with the possibility of using choice to represent the provisional choice associated with backtracking, which he calls “clairvoyant” choice. We described how we use semantic reversibility, encapsulated in the interaction between guards and choice and translated to executable statements, to achieve this backtracking automatically when confronted with infeasible choices.

We also showed, with some examples, the use of a construct based on the Prospective Values formalism which enables computations to be run and reversed, leaving no trace in the state except their result; this can be treated conceptually as the value that *would* result if the computation were to be run. An extension of this idea which utilises the backtracking interpretation allows the collection of a set of all possible results, for instance from a search, or to build sets of data satisfying various criteria. Our demonstration uses this in the context of the Sudoku puzzle to build sets of numbers available for blank squares, and using the guards and choice syntax to search for a route to a solution, whereby infeasibility triggers backtracking. The computational heart of the program (i.e. not input/output) was specified entirely in RB0, which was fully capable of expressing the operations required using abstract data structures and non-determinism while also being translatable into executable RVM code.

We looked next at the RVM itself, and the extensions to Forth which enable it to exploit virtual reversibility and implement Prospective Value computations, and also its capacity to use and manipulate abstract data types. Translations of a very simple program into C (generated by the B-Toolkit), and RVM code showed that the parameter-passing mechanism of the C version, with pointers and dereferencing, could be considerably simplified by using the RVM stack, though with the caveat that output variables now *must* be assigned. Another advantage of the RVM as an execution platform is that operations translated to conventionally coded RVM Forth can be invoked interactively, for piecewise exploration of their behaviour without necessarily running the entire program.

Translations of the Sudoku specification showed that the RB0 would generally map consistently and quite well to an RVM version; the question of whether it would be the most efficient we return to below. A deep and thorough analysis of the Sudoku process and the clues it provides to an experienced player would result in a program with a much reduced need for backtracking — perhaps even none at all. Rather than go down this route, we used a fairly naïve analysis with a single optimisation, and allowed the backtracking mechanism to take the strain. The resulting program still performs very creditably, solving “fiendish” puzzles in a fraction of a second, and backtracking up to several hundred times for varying degrees of difficulty.

To further increase the expressive power of RB0, we investigated extending the allowable scope of functions described by abstract constants to admit their use in RB0 implementations — as concrete constants. Such a facility would allow some computations to be concisely expressed, with recursion for instance; and the “black box” nature of the function leads to there being no changes of state to engender Invariant preservation proof obligations. We proposed a possible standardised syntax based on a combination of existing functional languages and Bunch theory syntax, to describe the output in terms of guarded bunches. The guards here must fulfil requirements of exhaustiveness and mutual exclusivity in order that the function have only one output. We showed that a syntax of this form is translatable into RVM; automatically, if well-formed.

A postfix version of Lambda calculus notation has been developed, and support for Lambda-style programs and techniques has been built into the RVM. We examined its possible use in RB0; this allows anonymous functions and the use of functions as first class objects — for instance their use as parameters and arguments to other functions, and closures. Some of this can currently be achieved at specification level in B, but must be refined away by the implementation stage; allowing its persistence to RB0 would introduce the possibility of powerful programming techniques from the world of functional programming. We have really only imperceptibly scratched the surface of what might be a useful addition to the armoury of B. While the question of translation was not considered in detail, the postfix Lambda notation was described; this might form the basis of an intermediate translation stage, carrying type information, using a two-pass compilation technique such as that described in Appendix G.

With regard to the translation of the mainstream operations in B, we provided preliminary schemas for the translation of RB0 and eventually RB1 into RVM. These schemas were presented based on RB0 (and thus ultimately on GSL), but AMN equivalents were provided where appropriate; these are semantically isomorphic to the corresponding GSL, so will map to the same RVM code. The schemas can be used to guide the actual translations when the syntax of RB1 is sufficiently mature; the most obvious candidate for future work arises in this section.

Bunch theory is a radical mathematical departure because, unlike Set theory, it treats collection and packaging as orthogonal activities. Owing to this radical nature, it has come in for some criticism, often implicit in the attitudes of some reviewers, who perhaps regard it as an undermining of the foundations of mathematics by researchers into programming theory. For this reason we explored some initial ideas relating Bunch theory to Set theory by a denotational semantics, thus returning to the foundations of the former. This allowed us to relate basic bunch axioms to analogous constructions in set theory, and prove bunch equivalences by translating both sides to arrive at the same set-theoretical representation.

9.1 Objectives

We now briefly revisit the list of objectives on page 3 in the Introduction.

1. The first objective was to review the physical background for our work and why it is a motivation; in chapter 3 we provided a brief history of Reversible Computing from initial concerns over heat dissipation through the realisation that viewing computations as reversible processes could, in theory, lead

to techniques to reduce this considerably. Another part of the objective was to consider broadly how the implications of these techniques could be incorporated into programming and specification; this was addressed in the second part of chapter 3, concluding that semantic reversibility was appropriate for a virtual machine, while a stepwise reversibility makes more sense at the hardware level.

2. The second objective was first to show how reversible constructs can be expressed in formal and programming terms; this was covered in chapters 4 and 5, with descriptions of the relevant features of RB0 and those of RVM. The use of Bunch theory in the analysis of the PV computations was described in chapter 4.

3. The third objective was to provide demonstrations and case studies showing development from RB0 to RVM-Forth. A number of small examples were given in chapter 4, with translations in 5, along with the Send More Money example (translation in Appendix E.) The larger case study of the Sudoku program and its translation also contributed to this objective. The development of RB1 is an area of future work arising from this, as included in the Future Work section below.

4. The fourth objective was to document the extensions and enhancements to RVM-Forth relevant to its support for reversibility in the Virtual Machine and abstract data structures; these were described in chapter 5. A linguistic summary is provided in Appendix B.

5. The fifth objective was concerned with how to allow abstract constants which defined functions to persist to implementation; in the first part of chapter 6 we addressed this by describing the current state of affairs and the problem of translation. We introduced a proposed standardised format with several examples of specification at RB0 level and translation to RVM.

6. The sixth objective involved ways of expanding the use of Lambda expressions and functions at implementation stage; this was addressed in the second part of chapter 6, wherein it was shown that a refined form of a lambda specification could (without its type information) be translated to a postfix form and thence to the RVM implementation, which was described in some detail. This is an area for future work, as some aspects of the RVM are still under development in order to ease the translation and ensure a reliable outcome for local and global variables.

7. The seventh objective was to provide translation schemas from the form of RB0 used in the thesis to RVM, and this was done in chapter 7. Future translation will concentrate on RB1 as the main source, with translation into a purer form of RB0 (for proof purposes) on the one hand, and RVM code on the other.

8. The final objective was to begin a process of formalising Bunch theory by denotational means, and a preliminary exploration of how this could be achieved is the subject of chapter 8. This is also an area for future work, with a deeper use of denotational concepts — those of environments and semantic functions, for instance — to provide verification of Bunch axioms.

9.2 Further Work

A major area of further work which emerges from the foregoing is the finalising of RB1 to a concrete syntax, which endeavour[1] will feed back to the final syntax of RB0. Then following the translation schemas for operations and functions, an actual automatic translation engine can be developed, and tested with RB1 versions of the included examples and others which have been developed over the last few years.

There is as yet no schema for Lambda calculus translation; this could be developed using the postfix lambda notation described in chapter 6 as an internal intermediate stage which can also carry type information, in order that the correct RVM forms of code are generated for its classes of data (e.g. floating point versus integer).

A few words on time are in order; we have not taken into account the question of non-termination of computations in the thesis, the assumption being that termination is assured for all examples. The B Method works within the total correctness framework, where the level of abstraction for time is "now or never". Formulations which include a fuller notion of time – one such being a calculus developed by Hehner – would be difficult for us to adopt. This is because our abstraction of backtracking uses something that might be called "clairvoyance", as if the computation could "see" from the start which paths were infeasible, and thus avoid them. Then these infeasible paths within a calculation are simply discarded, with no account being taken of the time required to explore them. The question of comparative evaluation of the complexity of reversible algorithms versus their irreversible counterparts is thus a matter for future research.

The two-pass compilation technique is likely to be of more general use in the translation of RB1, for instance using embedded "hints" with a specific syntax to automatically decide the most efficient representation of data types specified in terms of sets or sequences. While these are directly implemented in the RVM, for some purposes arrays might prove more efficient. This would impose certain restrictions on the available commands for sequences, should such a translation be the ultimate aim; some sequence operations are no more efficient in array form, and one or two are actually more trouble using arrays (for instance arrays are generally a fixed size, whereas a sequence can always be "grown". Extending a full array will be no easier than extending a sequence). The form of RB1 should also be translatable to RB0, as the former is intended to be the front-end specification environment.

[1]A PhD is already underway with this as one of its aims.

Another area for development is concerned with the question of automated proof facilities for RB0, comparable to those available for B. This endeavour will require deep knowledge of logic and the existing proof techniques used in B at present. A related enterprise was begun by Zeyda [31], but at the time the extent to which RB0 would differ from B0 was not known in detail; there is also the question of whether to base it around weakest precondition semantics, or the equivalent but more expressive Prospective Value semantics incorporating Bunch theory. The latter model seems more suited to the virtual reversibility environment, as the relevant constructs find readier expression therein.

A number of different but related areas of research will need to achieve a mature level of integration for this to be possible, the theory and praxis coming together in the production of a combined development environment involving RB1, its translation to RB0 for the discharging of proof obligations, and translation to RVM for executable code.

With regard to the RVM itself, it is currently a mature research tool, though work is underway to provide a more robust distributable form. Among the improvements will be a more user-friendly interface and debugging facilities, and ideally the RVM will be ported to other platforms — at least Windows, as it is currently confined to Linux. Versions which can be built from the assembler code and C (in the sets package) using current compilers will be required for both platforms, and also some form of API for its integration into a GUI-based RB-Tool environment.

Bibliography

[1] J-R Abrial. *The B Book.* Cambridge University Press, 1996.

[2] H G Baker. The thermodynamics of garbage collection. In Y Bekkers and Cohen J, editors, *Memory Management: Proc IWMM'92*, number 637 in Lecture Notes in Computer Science, 1992.

[3] C Bennett. The logical reversibility of computation. *IBM Journal of Research and Development*, 6, 1973.

[4] C Bennett. The thermodynamics of computation — a review. *International Journal of Theoretical Physics*, 1981.

[5] C Bennett. Notes on the history of reversible computation. *IBM Journal of Research and Development*, 32, 2000.

[6] E W Dijkstra. *A Discipline of Programming.* Prentice Hall, 1976.

[7] S E Dunne. Recasting the Hoare-He unifying theory of programs in the context of General Correctness. In A Butterfield and C Pahl, editors, *IWFM'01*, 2001.

[8] S E Dunne. A theory of generalised substitutions. In D Bert, J Bowen, M Henson, and K Robinson, editors, *ZB2002*, Lecture Notes in Computer Science, no 2272, 2002.

[9] R P Feynman. *Lectures on Computation.* Westview Press, 1996.

[10] M Frank. Reversible computing: Motivation, progress, and challenges. *Computing Frontiers 2005; ACM*, 2005.

[11] E Fredkin and T Toffoli. Conservative logic. Technical report, MIT Computer Science Lab, 1981.

[12] E Fredkin and T Toffoli. *Collision-based Computing.* Springer-Verlag, London, UK, 2002.

[13] E C R Hehner. Bunch theory: A simple set theory for computer science. *Information Processing Letters*, 12.1 pp26-31, 1981.

[14] E C R Hehner. *A Practical Theory of Programming.* Springer Verlag, 1993.

[15] R Landauer. Irreversibility and heat generated in the computing process. *IBM J R&D*, 5, 1961.

[16] A R Lynas and W J Stoddart. SuDoku Solver Case Study: from specification to RVM-Forth. In M. A. Ertl, editor, *21st EuroForth Conference Proceedings*, October 2005.

[17] A R Lynas and W J Stoddart. Adding Lambda Expressions to Forth. In M. A. Ertl and P. Knaggs, editors, *22nd EuroForth Conference Proceedings*, October 2007.

[18] G Michaelson. *Functional Programming through Lambda Calculus.* Addison-Wesley, 1989.

[19] J M Morris and A Bunkenburg. A theory of Bunches. *Acta Informatica*, 1999.

[20] J M Morris, A Bunkenburg, and M Tyrrell. Term transformers: A new approach to state. *ACM Trans. Program. Lang. Syst.*, 31(4), 2009.

[21] G Nelson. A generalization of Dijkstra's calculus. *ACM Trans. Program. Lang. Syst.*, 11(4), 1989.

[22] J Pöial. Typing tools for typeless stack languages. *22nd EuroForth Conference*, 2006.

[23] Science Daily. Easier route to 'spintronic' circuits? *Science Daily*, 2009.

[24] W Stoddart and P Knaggs. Type inference in stack based languages. *Formal Aspects of Computing*, 1992.

[25] W J Stoddart. An execution architecture for B-GSL. In Bowen J and Dunne S E, editors, *ZB2000*, number 1878 in Lecture Notes in Computer Science, 2000.

[26] W J Stoddart. Using Forth in an investigation into reversible computation. In P Knaggs and M A Ertl, editors, *19th EuroForth Conference*, 2003.

[27] W J Stoddart. Reversible virtual machine, user manual. Technical report, University of Teesside, 2005.

[28] W J Stoddart and F Zeyda. Implementing sets for reversible computation. In A Ertl, editor, *18th Euroforth, Technical University of Vienna*, 2002.

[29] W J Stoddart and F Zeyda. Expression transformers in B-GSL. In D Bert, J Bowen, S King, and M Walden, editors, *ZB2003*, Lecture Notes in Computer Science, no 2651, 2003.

[30] T Yokoyama and R Glück. A reversible programming language and its invertible self-interpreter. In *PEPM '07: Proceedings of the 2007 ACM SIGPLAN symposium*, pages 144–153. ACM Press, 2007.

[31] F Zeyda. *Reversible Computations in B.* PhD thesis, Teesside University, 2007.

[32] F Zeyda, W J Stoddart, and S E Dunne. The refinement of reversible computations. In T Muntean and K Sere, editors, *2nd International Workshop on Refinement of Critical Systems*, 2003. Available from www.esil.univ-mrs.fr/ spc/rcs03/rcs03.

[33] F Zeyda, W J Stoddart, and S E Dunne. A Prospective-Value semantics for the GSL. In H Treharne, S King, M Henson, and S Schneider, editors, *ZB2005*, Lecture Notes in Computer Science, no 3455, 2005.

[34] P Zuliani. Logical reversibility. *IBM J R&D*, 45(6), 2001.

Appendix A

Abstracts of Papers

SuDoku Solver Case Study: from specification to RVM-Forth

Angel Robert Lynas, Bill Stoddart

A project is underway to formulate a development cycle from B — suitably augmenting its implementation language B0 with reversibility constructs — to a coded implementation in the reversible target language RVM-Forth with translation schemas defined for this final stage. This paper describes the first phase of a case study using the puzzle SuDoku to investigate possible ways of fleshing out such a development cycle. We adopt an experimental approach, using a relatively simple specification as a springboard for what a generated code implementation might look like, and explore correspondences between the specification and implementation.

Adding Lambda Expressions to Forth

Angel Robert Lynas, Bill Stoddart

We examine the addition of Lambda expressions to Forth. We briefly review the Lambda calculus and introduce a postfix version of Lambda notation to guide our approach to a Forth implementation. The resulting implementation provides the basic facilities of an early binding functional language, allowing the treatment of functions as first-class objects, manipulation of anonymous functions, and closures.

Using Forth in a Concept-Oriented Computer Language Course

Angel Robert Lynas and Bill Stoddart

We describe a way of teaching fundamentals of Language Systems (for second-year Computing students), without having to compromise the use of a simple grammar owing to hardware limitations which need no longer apply in this setting.We adopt a top-down approach, reading from right-to-left and splitting the input string on the rightmost operator appropriate to that level. The target platform is Reversible Virtual Machine (RVM) Forth, so a postfix translation is the aim.

We introduce a basic arithmetic grammar and expand it during the course to allow unary minus, floating point and function application; this shows how type information can be generated in one pass and resolved in a second, via an internal intermediate code. Each version of the grammar has its productions mapped onto a system of equations which serve as the specification for the implementation functions.

A Virtual Machine for Supporting Reversible Probabilistic Guarded Command Languages

Bill Stoddart, Angel Robert Lynas, Frank Zeyda (University of York)

We describe a reversible stack-based virtual machine designed as an execution platform for a sequential programming language used in a formal development environment. We revoke Dijkstra's "law of the excluded miracle" to obtain a formal description of backtracking through the use of naked guarded commands and non-deterministic choice, with an operational interpretation of the interaction between guards and choice provided by reversibility. Other constructs supported by the machine provide for the collection of all results of a search, a semantically clean "cut" which terminates a search when the accumulated results satisfy some given criteria, and forms of probabilistic choice, which we distinguish from non-deterministic choice. The paper includes a number of example programs.

A Design-based Model of Reversible Computation

Bill Stoddart, Frank Zeyda, Angel Robert Lynas

We investigate, within the UTP framework of Hoare He Designs, the effect of seeing computation as an essentially reversible process. We describe the theoretical link between reversibility and the minimum power requirements of a computation, and we review Zuliani's work on Reversible Probabilistic Guarded Command Language. We propose an alternative formalisation of reversible computing which accomodates backtracking. To obtain a basic backtracking language able to search for a single result we exploit the already recognised properties of non-deterministic choice, using it as provisional choice rather than implementer's choice. We add a "prospective values" formalism which can describe programs that return all the possible results of a search, and we show how to formally describe the premature termination of such a search, a mechanism analogous to the "cut" of Prolog. An appendix describes some aspects of the wp calculus in terms of Designs, as needed for our proofs. Support for the programming structures described has been incorporated in a reversible virtual machine running under i386 Linux.

Popit: Implementing a Forth-like Language in C

Bill Stoddart, Angel Robert Lynas, Frank Zeyda

As programming environments and operating systems grow in complexity, students of computer science find it increasingly difficult to gain a holistic understanding of the computer systems they study. “Popit” is a Forth-like language designed to be implemented by students in order to expose them to a complete, though simple, programming system and virtual machine. The Popit implementation exercise is used in a group project module in the second year of the course, where it is implemented in C. In the fourth year of the course it is offered as an individual project to be implemented in Scheme.

Appendix B

Summary of RB0 and RVM

B.1 RB0 Notation

The RB0 used in the thesis is in a slightly hybrid state, as we have routinely used certain AMN-style constructs to aid readability — for instance IF-THEN rather than a decomposed guard/ choice pairing. The relationship between these was discussed in section 7.6.1, and such usages enable the reader to see the correspondence between the specification and the RVM code more clearly.

In chapter 7 we provided both versions where appropriate; we have eschewed the GSL $@z.S$ (unbounded choice for variables), along with two similar constructions for reasons explained in section 7.10; these all have the same translation in RVM, so we have used VAR-IN-END.

The usual mathematical notations of B are common to B and RB0, those used *in* substitutions. There follows a summary of the RB0 as used in the thesis, noting that some constructs would eventually be in the province of RB1. In this list, S and T are programs, G is a predicate, x is a variable of some type and A is a set of the same type as x.

Empty Operation. skip.

Assignment. Symbol :=, syntax $x := 1$.

Guard. Symbol $\Longrightarrow$, syntax $G \Longrightarrow S$, where G evaluates to true or false, and is generally assumed to have no side-effects.

Non-deterministic choice. Symbol [], syntax S [] T. The reversibility enables other choices to be taken until a feasible path is found.

Deterministic choice. **IF** G **THEN** S **END**, or **IF** G **THEN** S **ELSE** T **END**. These are non-reversing choices.

Assignment from Set. Symbol :∈, syntax $x :\in A$. Non-deterministically assigns x to some element of A.

Local variables. **VAR** x, y **IN** S **END**.

Sequential Composition. Symbol $\boxed{;}$, syntax *S* ; *T*. Execution of *S* followed by execution of *T*.

Prospective Values Computations Symbol ⋄, syntax $S \diamond E$. The value E would take if it were evaluated after executing S (implemented by executing S, evaluating E and storing the result, then reversing execution of E).

Loops. **WHILE** *G* **DO** *S* **END**.

B.2 RVM-Forth

There are a great many features and capabilities of ANSI Forth (and so the RVM) which are not our concern, as they relate to low-level programming, and the sort of constructs required to manipulate compilation to construct higher-level features. We have therefore concentrated on the features of the RVM used in relation to translation from RB0, and so directly relevant to our endeavour.

Also, the common mathematical operators are largely the same, allowing for conversion to postfix; these are as listed in sections 7.1.1 and 7.3. The elements below are those more related to substitutions.

The notation in this section will use italic letters S, T, U for program names (i.e. unspecified Forth operations), G, P for predicates (Forth expressions leaving true or false on the stack), and lower-case italic letters x, y, a, b, e for variable names. These can all be subscripted.

A definite value of any type (a literal expression or the contents of a variable) will be denoted by `V`, possibly numbered, and definite values of a *specific* type will be `I, $, S` or `P`, denoting an integer, string, set, or pair respectively.

Equality. RVM needs different symbols to represent the tests for equality in different classes of expression.

Integers	`I I =`
Strings	`$ $ STRING=`
Sets	`S S SET=`
Pairs	`P P PAIR=`

Variables. The declaration is

```
NULL VALUE x
NULL VALUE_ x
```

An underscore added to the keyword indicates the explicitly reversible version of a variable (reversible programs may not need all or any of their variables declared in this way). The variable can be initialised to a specific value by replacing `NULL` with the value.

An array declaration must specify the size *n* of the array:

```
n VALUE-ARRAY_ a
```

Assignments. The word `to` is used, as in

```
V to $x$
```

Multiple assignments can be performed, where several values are pushed onto the stack, and sequentially assigned.

```
V0 V1 V2 to $x_2$ to $x_1$ to $x_0$
```

Note that the values are assigned from the top of the stack, so the order of the variable names must reflect this.

Assignment to an element of an array will usually require that the index expression be enclosed in chevrons, to protect the contents from assignment by `to`. If a literal integer is supplied, the chevrons are not required.

```
V to << I >> of $a$
V to 3 of $a$
```

Selection (Non-backtracking). We have the two forms of the `IF` construction:

```
$G$ IF $S$ THEN
$G$ IF $S$ ELSE $T$ THEN
```

These can be nested, so either S or T could be another `IF`. Or multiple choices can be listed more intuitively with `CASE`. As this is non-backtracking, the final line is treated as an "otherwise" case, with `skip` if no action is specified.

```
CASE
$G_1$ ?OF $S_1$ ENDOF
$G_2$ ?OF $S_2$ ENDOF
⋮
$G_{n-1}$ ?OF $S_{n-1}$ ENDOF
$S_n$
ENDCASE
```

Non-deterministic Selection. Choice between programs is

```
<CHOICE $S_1$ [] $S_2$ [] ··· [] $S_n$ CHOICE>
```

Guarded choice:

```
<CHOICE
  $G_1$ --> $S_1$ []
  $G_2$ --> $S_2$ []
  ⋮
  $G_n$ --> $S_n$
CHOICE>
```

Loops.

```
        BEGIN G WHILE S REPEAT
```

Sets: Manipulation and Type Specification. There are some set manipulation facilities in RVM, the usual infix operations of union, intersection, and subtraction, plus domain and range extraction, being the most useful. Respectively, these are

```
S1 S2 \/      S1 S2 /\      S1 S2 \
S DOM      S RAN
```

In addition, another two words operate on sets to give the power set and cartesian product of two sets; these are `POW` and `PROD`, as in

```
S POW      S1 S2 PROD
```

These two words also occur with a somewhat different effect in the construction of a set; this requires that its type be specified, using combinations of the following words, each of which leaves a type specfier on the stack, or acts on those already there.

`INT`	Specifies integers.
`STRING`	Specifies strings.
`PROD`	The cartesian product of two sets, specifies maplets.
`POW`	Specifies a set of the preceding type specifier.

These are deployed in postfix as is usual; here, `POW` will operate on a type specifier to change it to a that of a set of that type, while `PROD` will operate on two type specifiers to produce the specifier for a maplet consisting of those classes.

So a specification like the following

```
STRING POW INT STRING PROD PROD
```

will allow the construction of a set with elements like (informally)

$$\{\text{ strings }\} \mapsto (\text{ integer } \mapsto \text{ string })$$

Set Construction. The subscript e_{TYPE} in the following few sections refers to the type of an element in the form of the type specifiers described above, which will provide the necessary information for the construction of a new set. The actual construction of a set begins, after the type has been specified, with the word $\boxed{\texttt{\{}}$.

$$e_{TYPE} \ \texttt{\{} \ e_1 \ \texttt{,} \ e_2 \ \texttt{,} \ \cdots \ e_n \ \texttt{,} \ \texttt{\}}$$

The comma is a word that allocates storage — so one is required after the final element.

Set Comprehension. A set comprehension (these are finite sets) uses the PV facilities to build a subset satisfying some predicate from a larger set.

```
S_TYPE { <RUN
  S CHOICE to x
  P --> x RUN> }
```

Non-deterministic Choice from a Set.

```
S CHOICE to x
S RANDOM-CHOICE to x
```

These will provide alternative choices on backtracking, while any remain.

Ordered Pairs. These can be constructed inside a set with the maplet operator `|->`, which will pick up the classes of the first and second element from the type specifier for the set. In other situations, the classes are symbolically suffixed to the operator to indicate this information. Constructing maplets of various class combinations will involve, for example

```
I $ |->I,$
$ S |->$,S
P S |->P,S
```

The available classes are integer, string, pair and set (suffix elements `I, $, S` or `P`), so there are sixteen possible combinations,

```
|->I,I  |->I,$  |->I,S  |->I,P
|->$,I  |->$,$  |->$,S  |->$,P
|->S,I  |->S,$  |->S,S  |->S,P
|->P,I  |->P,$  |->P,S  |->P,P
```

for the appropriate sequence of first and second element.

Sequences. The construction from a syntax point of view is essentially the same as for enumeration of sets, but using the sequence delimiters `[` and `]` which are of course Forth words in themselves.

e_{TYPE} `[` e_1 `,` e_2 `,` $\cdots$ e_n `,` `]`

Concatenation of two sequences M_i is

M_1 M_2 `^`

Appending has a built-in operator, but prepending requires the construction of a sequence with the new element, and its concatenation to the existing sequence.

M e `<-`
e_{TYPE} `[` e `,` `]` M `^`

Relational Image and Function Application. Where a relation R or function F is constructed as a set of maplets, there are operations to return the relational image (a set) or an element as a result of function application. In the following, $X \subseteq \operatorname{dom} R$.

> R X `IMAGE`
> F e `APPLY`

Prospective Value Computations. These have a non-deterministic application in set comprehension, above. Deterministic usage, where one result is expected, requires type specifiers in the closing delimiter, as below, to match the expression E. Thus there are four versions:

> `<RUN` S E `INT>`
> `<RUN` S E `STRING>`
> `<RUN` S E `PAIR>`
> `<RUN` S E `SET>`

We also include the collection of a single result from those available, non-deterministic and with garbage collection, $S \diamond_1 E$, which is

> E_{TYPE} `{ <COLLECT` S E `TILL CARD SATISFIED> } CHOICE`

Operation Definitions. There are a number of possible forms for these, depending on the requirement for a parameter list (for local variables initialised from the stack on invocation), or for internal local variables (see next section). We list the forms here, beginning with the simplest. The new operation is named S, and T is what it does.

> `:` S T `;`
> `:` S `(: :)` T `;`   (T has local variables, but parameter list empty)
> `:` S `(: VALUE_` x_1 `VALUE_` x_2 `...` `VALUE_` x_n `:)` T `;`

Local Variables. These are variables declared and initialised from inside an operation. They may or may not require a scope to be specified; if not, the delimiters can be omitted. The following clause can come anywhere after the parameter list (which must be supplied, even if empty).

> `(SCOPE 0 VALUE_` x_1 `0 VALUE_` x_2 `...` `0 VALUE_` x_n `SCOPE)`

Appendix C

Sudoku Listing

This is the complete listing for the Sudoku program developed and discussed in chapters 4 and 5. The first section initialises the data structure from a file read in by the startup command; some routines for printing out grids follow. The next sections are the computational heart of the program; finally we have the file read-in and running commands, along with a garbage-collecting wrapper for the whole program.

```
( ============ Initialisations ==================== )

NULL VALUE_ GRID  (  REV )
1 9 .. VALUE DIGIT
0 8 .. VALUE XY
( Following holds remaining CHOICEs for each blank)
NULL VALUE_ POSSIBLE  (  REV )
8 VALUE COLI 8 VALUE ROWI NULL VALUE_ ASSIGNED (  REV )

( Generalised GRID-builder; assumes file of 81 numbers loaded on stack)
: BUILD-GRID ( n TIMES 81 --  )
   8 to ROWI
   INT INT PROD INT PROD {
      BEGIN ROWI -1 >
      WHILE
         8 to COLI
         BEGIN COLI -1 >
         WHILE
            DUP 0= NOT
            IF ROWI COLI |->I,I SWAP |->P,I ,
            ELSE DROP
            THEN
            COLI 1- to COLI
         REPEAT
         ROWI 1- to ROWI
      REPEAT } to GRID ;
```

```
( ------------------------------------------------------------)
( Pretty(ish)-print subsystem. Can be ignored )

0 VALUE ELEMINDEX NULL VALUE GRIDSIZE

: UNPAIR ( x1.x2.* -- x1 x2 )
   DUP FIRST SWAP SECOND ;

: VLINE 124 EMIT ;
: LINE VLINE CR ." +---------+--------+---------+" CR  ;

( Convert co-ord pairs to scalar square numbers)
: CONVERTGRID ( n.n.*.n.P -- )
   INT INT PROD {
      DUP CARD 0 DO
         DUP I @ELEMENT
         UNPAIR SWAP UNPAIR SWAP 9 * +
         SWAP |->I,I ,
      LOOP
   } NIP ;

: .GRID ( n.n.*.n.P -- )
   CONVERTGRID
   0 to ELEMINDEX DUP CARD to GRIDSIZE
   81 0 DO ELEMINDEX GRIDSIZE < IF
         DUP ELEMINDEX @ELEMENT
      ELSE DUP 0 @ELEMENT
      THEN
      I 27 MOD 0=
      IF  LINE VLINE ELSE I 9 MOD 0=
         IF  VLINE CR VLINE ELSE I 3 MOD 0=
            IF VLINE
            THEN
         THEN
      THEN
      DUP FIRST I = IF
         SPACE SECOND . ELEMINDEX 1+ to ELEMINDEX
      ELSE 3 SPACES DROP THEN
   LOOP LINE DROP CR ;

( =========Utilties; zone generation etc ==============)

: GENROW ( n -- n.n.*.P )
   (: VALUE r  :)
   INT { r , } XY PROD ;

: GENCOL ( n -- n.n.*.P )
   (: VALUE c  :)
   XY  INT { c , } PROD ;

: CORNER ( n -- n )
     (: VALUE n  :)
   n n 3 MOD - ;

: SECTOR ( n n -- n.n.*.P )
   (: VALUE r VALUE c   :)
   r CORNER r CORNER 2 + ..
   c CORNER c CORNER 2 + .. PROD ;

( Find the constraint zone for a particular square )
: CZONE ( n n  -- n.n.*.P )
    (: VALUE r VALUE c   :)
  r  GENROW
  c GENCOL  \/
  r c SECTOR  \/    ;

( Now we find the values a blank square can take )
: AVAILABLE ( n n  -- n.P )
   (: VALUE r VALUE c   :)
   DIGIT GRID r c CZONE IMAGE \ ;

   ( -----------------------------------------------------------)

: INIT-POSSIBLE ( --  )
   (: :)  0 VALUE S 0 VALUE s 0 VALUE V
   XY XY PROD GRID DOM \ to S
   INT INT PROD INT POW PROD {
      <RUN
         S CHOICE to s
         s FIRST s SECOND AVAILABLE to V
         s V |->
       RUN> } to POSSIBLE ;

( Get the next squares from the most constrained -- returns a
  subset of POSSIBLE. We cheat a bit by using the set ordering
  to find the lowest card in the range of POSSIBLE )
: GETSQUARES ( -- n.n.*.n.P.*.P)
    (:  :)  0 VALUE n 0 VALUE x
   POSSIBLE DUP RAN ELEMENT CARD to n
   INT INT PROD INT POW PROD {
      <RUN CHOICE to x
            x SECOND CARD n = --> x
        RUN>  } 1LEAVE  ;

( Picks next square and its availables (an element of POSSIBLE)
  to send to assign )
: NEXTUP ( n.n.*.n.P.*.P -- n.n.*.n.P.*   )
     (: VALUE mset    :)   0 VALUE  x
       mset  PCHOICE to x  (  x .PAIR CR  )
       POSSIBLE x SUBTRACT-ELEMENT to POSSIBLE  x  ;

( Assigns from square and set of values a single value, adding
  pair to GRID. Hack to preserve square in nx )
: ASSIGN ( n.n.*.n.P.* --  )
   (: VALUE nx  :)   0 VALUE n
   nx FIRST VALUE SQ
   nx SECOND  CHOICE to n
   SQ n   |->P,I    to ASSIGNED ( Needs type for constructor )
   GRID ASSIGNED ADD-ELEMENT to GRID ;

( Remove assigned from the domain of each square in the constraint
  zone of the last assigned square -- update by func override. Should
  not attempt to update when no blanks need updating )
: UPDATE-POSSIBLE (  -- )
   (:  :)  0 VALUE U  0 VALUE upd  0 VALUE x
   ASSIGNED FIRST UNPAIR CZONE POSSIBLE <| to U
   U ?{} NOT IF
      INT INT PROD INT POW PROD { <RUN
         U CHOICE to x
         x FIRST  x SECOND ASSIGNED SECOND SUBTRACT-ELEMENT |->
       RUN> } to upd
       INT { } upd RAN IN NOT   -->
           POSSIBLE upd <+ to POSSIBLE THEN  ;

( ==========Run and step-through facilities ============= )

: START  ( -- )
   NULL to GRID NULL to POSSIBLE
   32 WORD LOAD-FILE BUILD-GRID INIT-POSSIBLE ;

: STEP  ( -- )
    GETSQUARES NEXTUP
    ASSIGN UPDATE-POSSIBLE  ;

: SOLVE ( -- n.n.*.n.*.P )
```

```
   BEGIN
      GRID CARD 81 <
   WHILE
      STEP
   REPEAT  ;

( Takes filename after word, e.g.: TRY-TO-SOLVE S1 )
: TRY-TO-SOLVE ( -- ) CR
   START SOLVE GRID .GRID   ;

( Garbage collecting wrapper for above )
.( TRY <filename> runs the puzzle )
: TRY
   <CHOICE
       <TRY TRY-TO-SOLVE CUT>
     []
       CR
   CHOICE> ;
```

Appendix D

Translating to C

This appendix has the small C translation example, introduced in chapter 2 and expanded in chapter 5; first a machine that specifies basic operations, then one that calls them. Then the implementation versions of these; the printouts were produced by the onboard LaTeX system in the B-Toolkit.

Following this, the C code generated by the B-Toolkit, and finally the RVM, a non-optimised version (with simplified variable names) based on the B0 versions.

```
MACHINE    $b2$

OPERATIONS

  $rr \longleftarrow$ VV ( $aa$ )   $\widehat{=}$
    PRE
      $aa \in \mathbb{N}$
    THEN
      IF   $aa > 4$   THEN
        $rr := 3 \times aa + 25 \times aa \times aa$
      END
    END

END
```

```
MACHINE    $b2s$

OPERATIONS

  $bb\ ,\ cc \longleftarrow$  SS ( $aa$ )  $\widehat{=}$
    PRE
      $aa \in \mathbb{N}$
    THEN
      IF   $aa > 4$   THEN
        $bb := 3 \times aa$  $\parallel$
        $cc := 5 \times aa$
      ELSE   $bb := 3 \times aa$
      END
    END   ;

  $yy \longleftarrow$  TT ( $xx$ )  $\widehat{=}$
    PRE
      $xx \in \mathbb{N}$
    THEN
      $yy := xx \times xx$
    END   ;

  $dd \longleftarrow$  UU ( $ee\ ,\ ff$ )  $\widehat{=}$
    PRE
      $ee \in \mathbb{N} \wedge ff \in \mathbb{N}$
    THEN
      $dd := ee + ff$
    END


END
```

IMPLEMENTATION *b2sI*

REFINES *b2s*

OPERATIONS

```
$bb$ , $cc \longleftarrow$ SS ( $aa$ ) $\widehat{=}$
    BEGIN
        IF $aa > 4$ THEN
            $bb := 3 \times aa$ ;
            $cc := 5 \times aa$
        ELSE
            $bb := 3 \times aa$
        END
    END ;
$yy \longleftarrow$ TT ( $xx$ ) $\widehat{=}$ $yy := xx \times xx$ ;
$dd \longleftarrow$ UU ( $ee$ , $ff$ ) $\widehat{=}$ $dd := ee + ff$
```

END

Cross-references for b2sI

b2s	MACHINE	1

IMPLEMENTATION *b2I*

REFINES *b2*

IMPORTS *b2s*

OPERATIONS

```
$rr \longleftarrow$ **VV** ( $aa$ )  $\widehat{=}$
    **BEGIN**
        **VAR**  $pp$ , $qq$ , $ss$  **IN**
            $pp$ , $qq \longleftarrow SS$ ( $aa$ ) **;**
            $ss \longleftarrow TT$ ( $qq$ ) **;**
            $rr \longleftarrow UU$ ( $pp$ , $ss$ )
        **END**
    **END**
```

Cross-references

SS	*b2s*	OPERATIONS	1
TT	*b2s*	OPERATIONS	1
UU	*b2s*	OPERATIONS	1

END

Cross-references for b2I

b2		MACHINE	1
b2s		MACHINE	1
SS	*b2s*	OPERATIONS	1
TT	*b2s*	OPERATIONS	1
UU	*b2s*	OPERATIONS	1

The translation to C produced by the B-Toolkit is:

```
//file b2s.h:
void INI_b2s(void);
void SS(int *_bb,int *_cc,int _aa);
void TT(int *_yy,int _xx);
void UU(int *_dd,int _ee,int _ff);

//file  b2.h
void INI_b2(void);
void VV(int *_rr,int _aa);

/* file b2s.c */

#include "b2s.h"   #
void INI_b2s(void)
{  ; }

void SS(int *_bb,int *_cc,int _aa)
{
  if ( _aa > 4 ) {
    *_bb = 3*_aa;
    *_cc = 5*_aa;
  }
  else {
    *_bb = 3*_aa;
  }
}
```

```
void TT(int *_yy,int _xx)
{
  *_yy = _xx*_xx;
} }
void
UU(int *_dd,int _ee,int _ff)
{
  *_dd = _ee+_ff;
}

/* file b2.c */

#include "b2.h"
#include "b2s.h" #
void INI_b2(void)
{
  INI_b2s();
}

void VV(int *_rr,int _aa)
{
  int pp,qq,ss;
  SS(&pp,&qq,_aa);
  TT(&ss,qq);
  UU(_rr,pp,ss);
}
```

And a fairly literal RVM translation from the implementation version, with parameter passing done via the stack.

```
: S ( n -- n n )
   (: VALUE a  :)
   0 VALUE b 0 VALUE c
   a 4 > IF
      a 3 * to b
      a 5 * to c
   ELSE
      a 3 * to b
   THEN  b c ;

: T ( n -- n )
   (:  VALUE x :)
   0 VALUE y
   x x * to y y ;

: U ( n n -- n )
   (:  VALUE e VALUE f :)
   0 VALUE d
   e f + to d d ;

: V ( n  --  n )
   (: VALUE a  :)
   0 VALUE r
   0 VALUE p 0 VALUE q 0 VALUE s
   a S to q to p
   q T to s
   p s  U to r r  ;
```

Appendix E

Send More Money

This is the complete RVM program to solve the Send More Money problem; after setting up some variables, the part based on the RB0 in figure 4.1 follows. Finally there is code to actually run the program and print out the result.

```
0 9 .. VALUE_ DIGIT
0 VALUE S ( -- n )   0 VALUE E ( -- n )   0 VALUE N ( -- n )
0 VALUE D ( -- n )   0 VALUE M ( -- n )   0 VALUE O ( -- n )
0 VALUE R ( -- n )   0 VALUE Y ( -- n )

: PUZZLE ( --, find a solution to the puzzle )
  (: :) ( empty list of named parameters )
    ( create local variables to represent the carry flags)
    0 VALUE c0  0 VALUE c1  0 VALUE c2
   1 to M  DIGIT INT { M , } \ to DIGIT ( make most constrained choice first)
    DIGIT CHOICE to D    DIGIT INT { D , } \ to DIGIT
    DIGIT CHOICE to E    DIGIT INT { E , } \ to DIGIT
    D E + 10 /MOD   to c0   to Y
    Y DIGIT IN  -->  DIGIT INT { Y , } \ to DIGIT
    DIGIT CHOICE to N   DIGIT INT { N , } \ to DIGIT
    DIGIT CHOICE to R
    N R + c0 + 10 MOD  E = -->
    N R + c0 + 10 / to c1
    DIGIT CHOICE to O
    E O +  c1 +  10 MOD  N =  -->
    E O +  c1 + 10 /  to c2
    DIGIT INT { R , } \ to DIGIT
    DIGIT CHOICE to S
    S M + c2 + 10 MOD  O =  -->
    S M + c2 + 10 / M = -->
  0LEAVE ;

: SOLUTIONS ( -- $.n.*.P.P, returns the set of solns to the puzzle,
each solution is in the form of a function from strings
(" S", " E" etc) to values. In fact there is only a single solution,
but we allow for multiple solutions in our approach )
  STRING INT PROD POW {
    <RUN PUZZLE
      STRING INT PROD {
        " S" S |-> ,  " E" E |-> ,  " N" N |-> ,  " D" D |-> ,
        " M" M |-> ,  " O" O |-> ,  " R" R |-> ,  " Y" Y |-> ,
      }
    SET>
  } ;

: .SOLN ( $.n.*.P -- ) (: VALUE SOLN :)
  SOLN " S" APPLY .  SOLN " E" APPLY .  SOLN " N" APPLY .  SOLN " D" APPLY .
  ." + "
  SOLN " M" APPLY .  SOLN " O" APPLY .  SOLN " R" APPLY .  SOLN " E" APPLY .
  ." = "
  SOLN " M" APPLY .  SOLN " O" APPLY .  SOLN " N" APPLY . SOLN " E" APPLY .
  SOLN " Y" APPLY .  0LEAVE  ;

: REPORT ( -- )
  (: VALUE SOLS :) CR
    SOLS CARD 0 DO
      SOLS I @ELEMENT .SOLN CR
    LOOP
  0LEAVE ;

: SENDMORY ( -- ) SOLUTIONS REPORT ;

CR .( SENDMORY runs the puzzle: SEND + MORE = MONEY )

SENDMORY
```

Appendix F

Reversible Pseudo-Random Number Generator

The code for an RNG which can repeat random numbers generated during a reversible program; past values are stored on its history stack, so can be restored with calls to `randpop`.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define SEED 500
#define BYTES 8
#define RANDSTACKSIZE 100
char randstate[BYTES];       // current random array state
char randstack[RANDSTACKSIZE][BYTES]; // history stack
// pointers for both
void * randstatep = randstate;
void * rstackp = randstack;

// push and pop functions
void randpush() {
   memcpy(rstackp, randstatep, BYTES);
   rstackp += BYTES;  // point at next free entry }

void randpop() {
   rstackp -= BYTES;  // point at last entry
   memcpy(randstatep, rstackp, BYTES);  }

// Wrapper to simplify calling rand
long revrandom() {
   randpush();
   return random(); }

//------------------ Demonstration Wrapper -----------------------
```

```
int main (int argc, char *argv[]) {
   initstate(SEED, randstate, BYTES);
   puts("Call random() a few times:");
   printf("\nInitially, R1: %ld, ", revrandom());
   printf("R2: %ld, ", revrandom());
   printf("R3: %ld\n", revrandom());
   puts("Reverse a single call...");
   randpop();
   printf("\nAnd we should get R3 again: %ld \n", revrandom());
   puts("Reverse twice to second val...");
   randpop(); randpop();
   printf("\nAnd now, R2: %ld, ", revrandom());
   printf("R3: %ld, ", revrandom());
   printf("R4: %ld\n", revrandom());
   return 0; }
======================================
A couple of sample runs.
 Run with 100 as SEED:
[u0019191@SCM-PC-633 12:25pm ~/progwork]$ a.out
Call random() a few times:
Initially, R1: 829870797, R2: 1533044610, R3: 1478614675
Reverse a single call...
And we should get R3 again: 1478614675
Reverse twice to second val...
And now, R2: 1533044610, R3: 1478614675, R4: 1357823696
------------------------------------------------------------
And with 500 as SEED:
[u0019191@SCM-PC-633 12:31pm ~/progwork]$ a.out
Call random() a few times:
Initially, R1: 2001820957, R2: 2037492626, R3: 479770979
Reverse a single call...
And we should get R3 again: 479770979
Reverse twice to second val...
And now, R2: 2037492626, R3: 479770979, R4: 991250784
```

Appendix G

Euroforth 2008

Using Forth in a Concept-Oriented Computer Language Course

Angel Robert Lynas and Bill Stoddart

University of Teesside
September 2008

Abstract

We describe a way of teaching fundamentals of Language Systems (for second-year Computing students), without having to compromise the use of a simple grammar owing to hardware limitations which need no longer apply in this setting.We adopt a top-down approach, reading from right-to-left and splitting the input string on the rightmost operator appropriate to that level. The target platform is Reversible Virtual Machine (RVM) Forth, so a postfix translation is the aim. We introduce a basic arithmetic grammar and expand it during the course to allow unary minus, floating point and function application; this shows how type information can be generated in one pass and resolved in a second, via an internal intermediate code. Each version of the grammar has its productions mapped onto a system of equations which serve as the specification for the implementation functions.

1 Introduction

In teaching Language Systems for Computer Science, the conceptual simplicities can often be obscured by the compromises made to accommodate purely historical restrictions on software writing for far more limited computers than those available today. These compromises take the form, for instance, of extra complications in the simple grammars, and trying to do several jobs in one pass of the compiler.

Accepting that we can now directly implement, at least in a pedagogical setting, a more straightforward approach less hampered by traditional constraints, can lead to a clarification of the conceptual issues involved in top-down parsing.

We used a basic grammar for infix arithmetical expressions, and used the conversion to postfix as a convenient route for exploring grammar analysis. An expansion to include floating point numbers and other facilities proved to be a fruitful method for introducing the usage of type information and two-pass compiling with intermediate code to hide the use of meta-information.

Following some definitions, we describe in Section 2 the overall approach taken, then in Section 3 the basic grammar presented to the students. In Section 4 we examine the extensions to this grammar, the actions of the new compiler's first pass and those of its second. We conclude with a brief look at a possible future application of these ideas for subsequent refinements of the course.

1.1 Some Definitions

In the following, we use some fundamental terminology from the study of grammars:

Terminals These are the strings which are the tokens of the language and appear in generated expressions.

Non-Terminals These are symbols which do not appear in the output, but stand for classes of terminals, or combinations of those classes.

Productions Specifies one way in which a non-terminal can be expanded to a sequence of other non-terminals and/or terminals, eventually generating strings composed entirely of terminals.

2 Higher-Level approach

The usual approach in teaching about language systems has been to adopt a relentlessly left-to-right method, compatible with historical restraints on memory, storage and processing capacity. Thus in the interests of efficiency, perhaps one token is available for "look-ahead" while the current token is being processed (look-ahead by default referring to the token on the right). See for instance the popular compiler texts [1] and [2].

Using these techniques, everything is done in a single pass, including storage and checking of type information (if applicable) and resolution of conditional structure — though we do not consider the latter here.

Certain things must be sacrificed to this methodology, the first casualty being simplicity of grammar. A grammar for basic arithmetic could contain a production (we define <expression> and <term> more fully later)

<expression>::=<expression> + <term>

meaning that one way of constructing an expression is by combining an expression and a term with a "+" sign in between. This we generally abbreviate to

$$E ::= E + T$$

This, however, is not really suitable for left-right parsing owing to the left recursion; that is, an E could expand to $E + T + T + \cdots + T$, so reliably telling where the first <expression> ends is problematical. The order of E and T specifies left-associativity for the + operator and others, ensuring that (for instance) $a - b - c$ is parsed as $(a - b) - c$, rather than $a - (b - c)$ which gives a different result.

In order to remove this left-recursion, such constructs are usually redefined using dashed letters to denote "the rest of the expression", now looking like this

$$E ::= TE'$$
$$E' ::= +TE' \mid \epsilon$$

The symbol ϵ or "*null*" stands for the empty string. The decomposition of the expression E can now be approached unambiguously from the left, as the term (processed by a similarly expanded rule) is recognised and the "rest of the expression" E' is passed downward for further processing. But this represents a considerable loss in simplicity compared to the first grammar.

What method, then, could be used to parse productions like $E \rightarrow E + T$ as is? Clearly a right to left approach would be more apropriate here, splitting the expression at the rightmost top-level "+" encountered, where the remaining expression on the left is dealt with by a recursive call to the expression parser, and the term is dealt with by the term parser. Terms are split on "*" or "/" if any occur, again the rightmost, for example $T = T * F$, where F is a factor with no top-level operations. We can now fill in the rest of the grammar and formalise the operation of a recursive parser.

3 A Simple Arithmetic Grammar

The initial grammar developed for the students, which allows for unsigned integers, identifiers, and bracketed expressions, is as follows. The order of the non-terminal expansions reflects the precedence order of the operators; lower precedence operators are scanned for first, and the non-terminal split if applicable. The higher precedence operators appear closest to the terminals in the postfix output and are thus executed first. The vertical bars on the left-hand side indicate alternative productions for the same non-terminal.

Non-Terminals:

E is an Expression; these can contain a plus or minus sign at the top level (i.e. not within any brackets).

T is a Term, which contains no pluses or minuses at its top level, but can contain "*" or "/" for multiply or divide.

F is a Factor, containing no top-level operations, but can expand to an unsigned number U, or an identifier I, or a bracketed expression (E), the contents of which are recursively expanded as an expression. We do not define U or I further at present; in any case U will be later replaced by a non-terminal which accommodates both integer and floating point numbers.

$$E ::= E + T \mid E - T \mid T$$
$$T ::= T * F \mid T/F \mid F$$
$$F ::= U \mid I \mid (E)$$

The compilation to postfix is described by a set of equations involving mutually recursive functions which act on appropriate strings of each non-terminal class. The right-hand sides of these equations show the output from a given string; at the top levels, this will involve invoking other functions on part of the string

— possibly including the function itself recursively. In the following, lower-case e, t, f, u and i represent strings belonging to the nonterminals E, T, F, U and I respectively; that is, e is a particular expression, t is a particular term, and so on. We have the functions

P_E takes a string e and returns the translation of that string in postfix.

P_T takes a string t and returns the translation of that string in postfix.

P_F takes a string f and returns the translation of that string in postfix.

If no operators are found by P_E or P_T, they pass the entire string down to the next function. The symbol "⌒" is used for string concatenation. The mutually recursive equations which define the compilation are

$$P_E(e \frown \texttt{"+"} \frown t) = P_E(e) \frown P_T(t) \frown \texttt{" +"} \tag{3.1}$$

$$P_E(e \frown \texttt{"-"} \frown t) = P_E(e) \frown P_T(t) \frown \texttt{" -"}$$

$$P_E(t) = P_T(t)$$

$$P_T(t \frown \texttt{"*"} \frown f) = P_T(t) \frown P_F(f) \frown \texttt{" *"} \tag{3.2}$$

$$P_T(t \frown \texttt{"/"} \frown f) = P_T(t) \frown P_F(f) \frown \texttt{" /"}$$

$$P_T(f) = P_F(f)$$

$$P_F(u) = u$$

$$P_F(i) = i$$

$$P_F(\texttt{"("} \frown e \frown \texttt{")"}) = P_E(e)$$

At the top two levels — expression and term — the parsing is right to left; specifically, a Forth operation `LSPLIT` is used to search for a symbol in $\{+, -\}$ or $\{*, /\}$ at the top level. The stack parameters are the string to be searched and a sequence of strings which are the tokens to search for. An example call follows, but a brief explanation is needed first.

A string in RVM-Forth can be an "ASCII-Zero" or AZ string, which is terminated by an ASCII null (in other words, a 0 after the final character, like strings in C and some other languages). They are created with the $\boxed{\texttt{"}}$ word, terminated by another quote. Also, sequences are created with the syntax

```
<type> [ <element1> , <element2> , ]  ( and so on)
```

where the comma word allocates storage for each element. The type can be simple (integer, string) or composite, involving pairs and nested sets or other sequences.

An example call to `LSPLIT` using a string `EXPR1`, then, is

```
EXPR1 STRING [ " +" , " -" , ] LSPLIT
```

This would be an call to process `EXPR1` using the symbols $\{+, -\}$.

Should `LSPLIT` encounter a right bracket ")", this means that a lower-level expression is included in the top-level one, which expression will be dealt with by a later recursive call to P_E. So `LSPLIT` will not continue its search for operators until it finds the matching left bracket; it will keep track of the level, incrementing for any right brackets and decrementing for matching left ones, until its own level zero is reached again.

On encountering a symbol/token in its search set, the operation splits the input string into a left part, the symbol string, and a right part. As can be seen from equations (3.1) and (3.2) above, the first part is sent recursively to P_E or P_T as appropriate, the second part to P_T or P_F, while the operation token is output last of all, so that it occurs *after* the outputs from parsing the rest of the string. If no search symbol is found, the entire string is passed down to the next parsing level.

When the factor level is reached, subsequent processing of U and I can be done left to right, as no left recursion occurs in these definitions. A factor of the form "(E)" merely has its brackets stripped and the contents sent back to the top-level function P_E. We include the code for P_T (called `PT`), which parses terms into terms and factors, as figure 1 below. The local variable syntax uses the words `(:` and `:)` to delineate the declarations, with values being taken from the stack.

```
: PT ( az1 -- az2, parse a term, leaving az2 the postfix
       translation of the term az1 )
   (: VALUE e :)
    e STRING [ " *" , " /" , ] LSPLIT
    VALUE BEFORE VALUE AFTER VALUE OP-STRING
    OP-STRING NULL =
    IF
      BEFORE PF      ( No op found, so e was a factor )
    ELSE
      BEFORE RECURSE  AFTER PF  ^  OP-STRING  ^
    THEN
  1LEAVE ;
```

FIGURE 1: RVM-Forth code for P_T.

The general technique is known as Recursive Descent Parsing, and is well-known, though we do not know of its having been implemented in this bidirectional way. The usual categories of LL(k) or LR(k) do not apply, strictly speaking, as they denote solely unidirectional parsing; for instance the Earley recogniser [3], based on Knuth's LR(k) algorithm. Our technique is adaptable insofar as right-associative operators can and will be accommodated by a sort of mirror image of `LSPLIT` which would read from left to right.

The parsing can be illustrated with a couple of examples, which can either be represented as parts of trees (useful for the students initially), or in a linear fashion which follows the equations closely. Given the input string "$x{*}x{-}1{-}(x{-}1){*}(x{+}1)$", for instance, the first parsing step is shown in figure 2.

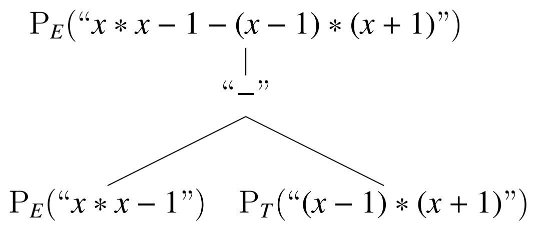

FIGURE 2: Splitting at the top level.

The left-hand side of this is then the input to a recursive call to P_E; the rest of this is shown (as far as the Factor level) in figure 3.

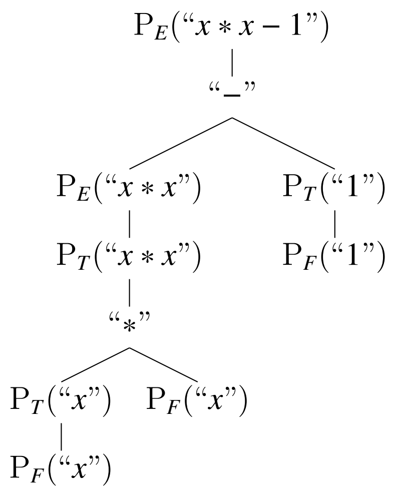

FIGURE 3: Splitting the left-hand half down to Factor level.

In linear style, the first part of this second tree would appear as

$$P_E(\text{"}x * x - 1\text{"}) = P_E(\text{"}x * x\text{"}) \frown P_T(\text{"}1\text{"}) \frown \text{" - "}$$

which we can see follows the pattern of equation 3.1.

As regards performance, we have not yet undertaken any exhaustive time complexity analyses. Some empirical run-throughs indicate that the basic technique is probably $O(n^2)$ for a suitable grammar; the limitations with respect to which grammars can be handled require further investigation, however.

4 Floating Point Extension

We now consider extending the grammar to include unary minus, function application and floating point capability. The inclusion of mixed-mode arithmetic requires the simple compiler to become a two-pass compiler, whereby the first pass generates intermediate code containing type information, as we now have integer and floating point to deal with. These require different operations for arithmetic and printing, and sometimes conversion is required from integer to floating point.

Unary minus is dealt with by converting the minus sign into a tilde "∼", as the compiler uses this internally; it is converted back for the Forth-readable output, and allows multiple minuses in a row which are converted consistently. The details are not examined further here. Function application is of the form "<identifier>(< arg-list>)", with the arguments comma-separated.

The basic grammar is extended as follows, with the extra non-terminals

F_0 This is simply an unsigned factor.

N This replaces U, and is a general number (integer or float), parsed by the floating point state machine to be described later.

L This is a comma-separated list of arguments to a function.

$$\begin{aligned}
E &::= E + T \mid E - T \mid T \\
T &::= T * F \mid T/F \mid F \\
F &::= F_0 \mid -F \\
F_0 &::= N \mid I \mid (E) \mid I(L) \\
L &::= L, E
\end{aligned}$$

The list of arguments L is parsed right to left, in a similar way to E and T, but split by `LSPLIT` on a comma.

4.1 First Pass (intermediate code)

The generating functions for the upper levels are similar for those in the previous section, but instead of including a straightforward plus or multiply sign in the output, they include the string for an internal operation which the second pass of the compiler will use to resolve the types and assign the correct integer or floating point version of the Forth operation in the final output; these internal operations all have an underscore suffix. The first few equations are thus similar to those near (3.1).

$$P_E(e \frown \texttt{"+"} \frown t) = P_E(e) \frown P_T(t) \frown \texttt{" +\_"}$$
$$P_E(t) = P_T(t)$$
$$P_E(e \frown \texttt{"-"} \frown t) = P_E(e) \frown P_T(t) \frown \texttt{" -\_"}$$
$$P_T(t \frown \texttt{"*"} \frown f) = P_T(t) \frown P_F(f) \frown \texttt{" *\_"}$$
$$P_T(t \frown \texttt{"/"} \frown f) = P_T(t) \frown P_F(f) \frown \texttt{" /\_"}$$
$$P_T(f) = P_F(f)$$
$$P_F(\texttt{"("} \frown e \frown \texttt{")"}) = P_E(e)$$

Below this, terminals begin to be output in the intermediate code, however these need to be strings rather than literals; spaces are added where required to ensure the final output is consistently space-separated. In the equations below, the symbol $\boxed{\texttt{\textbackslash"}}$ denotes a literal embedded quote, and ϕ is a function identifier (not a numerical variable). We also have the additional parameters

- n Any integer number string.
- i Any integer identifier (variable) string — note, no longer a general identifier
- l Any list of arguments.
- r Any floating point number string.
- r' Any Forth equivalent[1] to r.
- x Any real identifier string (floating point variable).

The identifiers are stored in symbol tables which specify some predefined families of identifier strings; this obviates any immediate need for typed declarations, however this is an area which will be examined at a later stage. Thus identifiers beginning with the letters `i-n`, in either case, are integer while all others are treated as floating point.

We then introduce the additional parsing functions

- P_{F0} Returns the intermediate code for an unsigned factor.
- P_L Returns the intermediate code for an argument list.

[1]We distinguish these as they may use different symbols for unary minus in exponent notation.

The remaining conversion equations for the intermediate code can now be defined (note L is shorthand for l, e; and l could be a single e).

$$P_F(\text{" } \sim \text{ "} \frown f) = P_F(f) \frown \text{" }\texttt{NEGATE\_"}$$
$$P_F(f) = P_{F0}(f) \qquad (f \text{ can be } n, i, r, x, \text{ or } \phi)$$
$$P_{F0}(\phi(L)) = P_L(L) \frown P_{F0}(\phi) \qquad (P_{F0} \text{ would simply output the string } \phi)$$
$$P_{F0}(n) = \backslash\text{"} \frown n \frown \backslash\text{"} \frown \text{" }\texttt{int"}$$
$$P_{F0}(i) = \backslash\text{"} \frown i \frown \backslash\text{"} \frown \text{" }\texttt{int"}$$
$$P_{F0}(r) = \backslash\text{"} \frown r' \frown \backslash\text{"} \frown \text{" }\texttt{float"}$$
$$P_{F0}(x) = \backslash\text{"} \frown x \frown \backslash\text{"} \frown \text{" }\texttt{float"}$$
$$P_L(l \frown \text{","} \frown e) = P_L(l) \frown P_E(e)$$
$$P_L(e) = P_E(e)$$

The final four P_{FO} equations show the type information being included in the output string; for example, the tokens "36" or "3.142" become the strings (with embedded quotes around the numbers)

" " 36" int"
" " 3.142" float"

This is because we are still in the intermediate code, which will be converted to input for Forth by the second-pass operations — those with the appended underscore — e.g. `+_` , `*_` . We can now examine these in the context of the second pass of the compiler.

4.2 Second Pass

For the simple example infix input $10.5 + 5 * 2.5$, the first pass will have produced the intermediate output string with embedded quotes

```
" 10.5" float " 5" int " 2.5" float *_ +_                    (4.1)
```

This is then tokenised and interpreted by the second-pass compiler. At this level, `int` and `float` are interpreted as integer constants, so when the operation `*_` is reached, the stack has three string/integer pairs on it. The code for `*_` is shown in figure 4.

It takes four arguments consisting of two string/integer pairs, the integers containing type information. The four possible combinations of `int` and `float` are checked (the second case is applicable here). The numerical/ variable values are output as strings, followed by the appropriate ordinary arithmetical operators, here either `*` or `F*` (the floating point version). In addition, since `F*` requires

```
: *_  ( az1 type1 az2 type2 -- az3 type3)
 (: VALUE e1  VALUE type1  VALUE e2  VALUE type2 :)
   CASE
     type1 int =  type2 int =  AND  ?OF
       e1 e2 ^ " *" ^ int
     ENDOF
     type1 int =  type2 float =  AND  ?OF
       e1 " S>F" ^ e2 ^  " F*" ^  float
     ENDOF
     type1 float =  type2 int =  AND  ?OF
       e1  e2 " S>F" ^ ^  " F*" ^  float
     ENDOF
     type1 float =  type2 float =  AND  ?OF
       e1  e2  ^  " F*" ^  float
     ENDOF
     ( otherwise ) " Type error" AZ-ABORT
   ENDCASE
 2LEAVE ;
```

FIGURE 4: RVM-Forth code for `*_` .

that both its arguments be floating point, the conversion operator[2] `S>F` is inserted in the output string after any integer values.

Finally the constant corresponding to the value is output, either `int` or `float`. Thus the output is a string and an integer, suitable for input to another operation of this kind.

After `*_` has interpreted and dealt with the relevant parts of string (4.1), we come to `+_` , which finds the following four arguments on the stack:

```
"10.5" float "5 S>F 2.5 F*" float
```

the last two being the output from `*_` . Note these strings are actual strings.

The code for `+_` is very similar to the above, and the final output will be

```
"10.5 5 S>F 2.5 F* F+" float
```

The final type indicator is dropped from this (by the second-pass compiler when the end of the expression is reached), and the remaining string can now be compiled by Forth.

For an example with identifiers (subject to the conventions mentioned in 4.1, page 8) we use

$$(i + 7) * (j + 1.5)$$

The first pass produces the *string* (recall these quotes are embedded)

[2]Single-precision integer to Floating point.

```
" i" int " 7" int +_ " j" int " 1.5" float +_ *_
```

After both `+_` have been processed, the stack will contain the following:

```
"i 7 +" int "j S>F 1.5 F+" float
```

Finally `*_` will produce this

```
"i 7 + S>F j S>F 1.5 F+ F*"
```

having dropped the `float` type indicator.

The second pass does not have such a neatly defined set of equations to encapsulate it, as the arguments are no longer a single string, but four arguments (stack parameters): string, integer, string, integer. Also, `NEGATE_` and `F->F_`, dealing with negating variables and numbers, and type-checking the arguments for function applications[3], have different signatures again. They also have more conditions to cover with, for instance, four combinations of `int` and `float`. The specifying equations for `+_` would be:

$$\texttt{+\_}(n_1, \texttt{int}, n_2, \texttt{int}) = n_1 \frown n_2 \frown \texttt{" +"}$$

$$\texttt{+\_}(n, \texttt{int}, x, \texttt{float}) = n \frown \texttt{" S>F"} \frown x \frown \texttt{" F+"}$$

$$\texttt{+\_}(x, \texttt{float}, n, \texttt{int}) = \texttt{" S>F"} \frown x \frown n \frown \texttt{" F+"}$$

$$\texttt{+\_}(x, \texttt{float}, x, \texttt{float}) = \texttt{" S>F"} \frown x \frown \texttt{" S>F"} \frown x \frown \texttt{" F+"}$$

4.3 Floating Point State Machine

For the general numeric literals subsystem — which handles floating point and exponent form literals — we adopt a state machine. A feature of the exponent form for infix expressions is that unary minus in an *exponent* must be written as a tilde rather than a normal minus (this does not apply to any other unary minus). Thus expressions such as

`-3.467e~6` or `-2.5e~10`

are admissible. This is to simplify the otherwise unwieldy process of identifying unary minuses for floating-point exponents and replacing them internally with tildes; the outputs will have normal minus signs for the usual Forth input. The machine itself has seven states; N_2, N_4 and N_7 are terminal states, and ϵ denotes an empty string.

[3]Neither are described in detail here.

$N ::= N_1 \mid DN_2 \mid .N_3$

$N_1 ::= DN_2 \mid .N_3$

$N_2 ::= \epsilon \mid DN_2 \mid .N_4 \mid eN_5$

$N_3 ::= DN_4$

$N_4 ::= \epsilon \mid DN_4 \mid eN_5$

$N_5 ::= N_6 \mid DN_7$

$N_6 ::= DN_7$

$N_7 ::= \epsilon \mid DN_7$

5 Conclusions and Further Work

The idea for this course stemmed from a desire to teach Language Systems at a more pure conceptual level, so that the concepts would be less obscured by their almost immediate abandoning for a more complex bottom-up approach, as was previously done.

We therefore used a top-down right-to-left method to parse expressions down to factor level, using recursive descent techniques to generate postfix versions readable by the Forth system — later via intermediate code to hold type information. The course thus began with deceptively simple concepts and could build naturally to a reasonably capable two-pass expression compiler. So the students learn about grammars and programming simple compilers in Forth.

We could go in a number of different directions with this, perhaps expanding in other ways than the floating point facility; for instance, one idea is to adapt this approach to make use of RVM Forth's native support for sets (using the C package contributed by Frank Zeyda [5]). This could be used to develop, for the course, an application to accept sets and basic set operations and generators from a suitably defined grammar, and convert them to valid Forth. Even eschewing floating-point support here, we still have the issue of types, as sets can contain strings and integers, and also pairs (maplets, using the ASCII notation `|->`) and nested sets. Therefore recursion would again be required to tease out the type information at each level and generate the appropriate tags for the first-pass output.

For illustration, a brief example of a simple set enumeration with string-integer pairs and the output which would be generated:

```
{"Dave" |-> 3291, "Li" |-> 3419} becomes
STRING INT PROD { " Dave" 3291 |-> , " Li" 3419 |-> , }
```

The grammar would, among other things, have to ensure that the maplet operator retained left associativity, and had the correct type signature. Some operators have right associativity, for instance domain restriction, and the bidirectional functionality will be needed to parse these. We will give consideration to a generalised and consistent system for type declarations in these language grammars, which would work with the set types of Forth and also with numeric types; also a closer analysis of the time complexity over a greater variety of grammars.

References

[1] A. V. Aho and J. D. et al Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 2nd edition, 2006.

[2] R. C. Backhouse. *Syntax of Programming Languages: Theory and Practice.* Prentice-Hall, 1979.

[3] J. Earley. An Efficient Context-Free Parsing Algorithm. *ACM, Vol 13 no. 2*, 1970.

[4] W. J. Stoddart and A. R. Lynas. A Reversible Computing Approach to Forth Floating Point. In M. A. Ertl and P. Knaggs, editors, *23rd EuroForth Conference Proceedings*, October 2007. On-line proceedings.

[5] W. J. Stoddart and F. Zeyda. Implementing Sets for Reversible Computation. In M. A. Ertl, editor, *18th EuroForth Conference Proceedings*, 2007. On-line proceedings.

The RVM-Forth software and manual can be downloaded from

```
http://www.scm.tees.ac.uk/formalmethods/download/rvm0_1.zip
http://www.scm.tees.ac.uk/formalmethods/download/rvmman.pdf
```