Update Plans

A High Level Low Level Specification Language

Hugh Osborne

Department of Informatics
Faculty of Mathematics and
Informatics
University of Nijmegen
Toernooiveld
6525 ED Nijmegen
The Netherlands

Table of Contents

Preface

This thesis is the result of work carried out between 1989 and 1994. During the first two years of this period I was a research assistant, at the University of Nijmegen, in Esprit Basic Research Action no. 3147 (the Phœnix project), carried out in collaboration with Imperial College in London and GMD in Karlsruhe, and financed by the European Community. The remainder of the period I was an "AIO" (postgraduate research assistant) at the University of Nijmegen. I am grateful to both the European Community and the University of Nijmegen (and thus, indirectly, to the Dutch and European tax payers) for financing this research.

Some of the work in this thesis has been presented earlier. In particular chapters two, three and four are based on work presented in [56, 57, 58, 59, 60].

I would like to thank the many people who have contributed to the development of the ideas presented in this thesis. First and foremost, I would like to thank Hans Meijer, who originally developed Update Plans (then called Update Schemes) [53], and who has played a fundamental rôle in helping to develop and formalise the key concepts in this thesis. I would also like to thank Robert Giegerich, of the University of Bielefeld, whose ideas formed the inspiration for chapter eight and, indirectly, chapters seven and six. I would, of course, also like to thank my colleagues on the Phœnix project, and in particular Hendrick Lock (GMD, Karlsruhe), whose JUMP machine [47, 48] formed one of the first test cases for Update Plans, and Erik Meijer (KUN, Nijmegen) and Ross Patterson (Imperial, London), whose λMMachine [52] is specified in chapter four. Other people I would like to thank are Mark van den Brand, Todd Cook, Max Geerling, Wil Lamain, Kees Koster, Ineke Kusters, Greta Löw and Harrie van Seters. The list would, of course, not be complete without mentioning Mariël, Sara and Tobias. Their patience and encouragement ("Papa, wanneer is jouw boekje nou af?" [66]) were essential to the completion of this work. Sara deserves a special mention for designing the cover.

Chapter One Introduction

It has been said [38] that computer programming is an art, and this is undoubtedly true — if the products of civil engineers had to be "debugged" as often as those of software engineers do, the world would be a hazardous place. Besides threatening to banish computer science to a no-man's land between C.P. Snow's two cultures [69], this lack of precision has more serious consequences. As computer systems become ubiquitous, the software engineer's products begin to play as important a part in daily life as do the civil engineer's. One of the major tasks, therefore, of computer science — perhaps the subject's whole *raison d'être* — is to supply formalisms and methodologies for constructing provably correct software. Historically this problem has been addressed on three fronts.

Methodologies have been invented for the development of software by the transformation of precise but conceptually simple specifications of problems to less intuitive but more easily implementable programmes [5, 7, 8, 9, 50, 51, 62, 70, 71].

Programming languages have become more and more precisely defined [54, 75] making precise definitions of the semantics of programmes more and more feasible.

Compilers are of course themselves programmes. By a combination of these two techniques formal specifications of languages are transformed to working compilers [20, 25, 33, 34, 43, 53].

These three methods all address the problem of translating or compiling problem specifications in some high level language to an equivalent specification in a low level language, mainly by means of syntactic transformations. The semantics of the low level language are usually not (formally) specified, but assumed to be trivial. This is an unwarranted assumption.

Example 1

This classic example, from the architecture of the PDP-11, shows the dangers of assuming the semantics of low level languages to be trivial. Consider the predecrement and postincrement addressing modes. The semantics seem simple. In the predecrement mode an address in a register is decreased by one (byte or word) and used as the address of the operand, e.g. if register `R6` contains the address `1234` then the object accessed by `-(R6)` is the object at address `1233`, and this is also the address that will be found in `R6` after the access. The postincrement mode is the inverse, first the object is accessed and then the address is increased. Addressing mode `(R6)+` will address the object at address `1234`, after which `R6` will contain `1235`.

A formal specification of the semantics of these addressing modes seems superfluous. In combination, however, they are less simple. Take, for example, the instruction '`MOV (R6)+ -(R6)`'. What does it mean? Is the postincrement executed, and then the predecrement, or is the order of access random? Maybe the arguments are accessed simultaneously, but what does this mean? Perhaps a formal specification is necessary after all.

Formal specifications, derived using the methods above, cannot be considered complete without attention to the details of the semantics of the underlying low level language — usually the instruction set of some machine, either concrete or abstract. A formalism is needed in which the semantics of such low level languages can be specified. Providing an important link in formal specification methods is not the only use of such a formalism. In the case of specifications of abstract machines, or concrete machines under development, the specification can be useful in reaching design decisions, and optimising the instruction set. In the case of existing concrete machines it can be used for deriving provably correct optimisations of generated code.

Update Plans are intended as a well defined specification formalism, with sufficient abstractive power to facilitate specification, yet close enough to concrete machine models to enable simple implementation of specification based prototypes.

1 Update Plans

Update Plans are a form of rewrite system, in particular a graph rewrite system [10, 21], closely related to term graph rewriting [6, 72] (though this will not be pursued in this thesis). They combine a well defined formal semantics, and an intuitive operational semantics with readability and flexibility. They were were introduced by Meijer [53] to describe machine-code generated by a compiler in a machine independent way. They were deliberately designed in such a way that an update plan resembles a set of rewrite rules or, in a sugared version of the syntax, function definitions in

Traditional array addressing

Update Plan addressing

Figure 1: Addressing in arrays and update plans

a declarative language.

Update plans specify state transitions in some abstract machine. This machine consists of a number of *stores*, each containing a countable set of *cells* addressed by a completely ordered set of locations, called *locators*. Conceptually it is not the cells themselves that are addressed, but the boundaries between cells, so that a sequence is not considered to be *at* certain locators, but rather *between* locators. For example, rather than representing the string 'CAT' by an array of characters, and considering the substring 'AT' to be the subarray from locator 2 to locator 3, 'CAT' is considered to be a sequence of characters between locator 0 and locator 3, and 'AT' to be the subsequence between locators 1 and 3. This is illustrated in figure 1. The basic notation for such a sequence is `0['CAT']3`. An extensive motivation for this numbering convention is given by Meijer [53]. Suffice it here to mention that `a[x y]c` is the same as `a[x]b b[y]c`, for some `b`, and that with `a[x]b` we have length `x` $=$ `b` $-$ `a`.

The notation `a+n` is used to refer to the locator n cells to the right of `a`, and `a−n` to refer to the cell n cells to the left. A singleton sequence is identified with its element. Sequences are specified by triples of the form

$$\langle locator \rangle\, [\, \langle sequence \rangle\,]\, \langle locator \rangle\,,$$

called *locator expressions*. A grammar for basic Update Plans is given in chapter two. A set of locator expressions is *consistent* if there are no two expressions in the set which specify the contents of some cell to have two different values. A consistent set of locator expressions describes a (sub)configuration of the machine.

The immediate constituent part of an update plan, an *update scheme*, is constructed from two sets of locator expressions forming the *left hand side* and the *right hand side*, and from a boolean expression, known as the *guard*. Variables (indicated by lower case words) are allowed in the constituent expressions of an update scheme. An update scheme containing only constants (indicated either by a value or, symbolically, by upper case words) is known as an *update rule*. Some simple examples can be found later in this chapter. Given a substitution and evaluation mechanism mapping variables to (sequences of) constants an update scheme can be instantiated to an update rule. Both the left and right hand sides of the resulting update rule must be self-consistent, i.e. all locator expressions in the left and right hand side respectively must be mutually consistent. The left and right hand side, however, need not be consistent with each other.

Briefly, Update Plans work as follows (more detailed descriptions are to be found in the remainder of this thesis). An update rule is *applicable* to a given configuration if its left hand side is a subset of that configuration and its guard is true. The result of applying an applicable update rule to a configuration c is the superset of the right hand side that is minimally different from c. A set of update schemes is called an *update plan*. One configuration can be *derived* from another (source) configuration by an update plan if the plan contains an update scheme an instantiation of which is applicable to the configuration. The derived configuration is then the result of applying such an instantiation to the source configuration. Configurations can also be derived by repeated applications of an update plan. A *final configuration* for an update plan is a configuration to which none of the update schemes in the plan is applicable. A *script* is an update plan together with an *initial configuration*. A *final development* of a script is a final configuration derived from the script's initial configuration by applications of the script's update plan.

A glossary of the terminology of Update Plans can be found in appendix VIII. A formal definition of the syntax is given in chapter two, and of the semantics in chapter three.

2 Related Approaches

Various other methods have been proposed for specifying low level activities. Abstract machines have been specified using transition systems [22], informal descriptions [73], an imperative programming style [47, 63] and functional languages [41], to name but a few. The best known contributions from the concrete side are probably ISPS [68] and register transfer languages, for example three address code [1]. These methods all have their drawbacks, lacking either a formal definition, or being impractical in that they are not easy to realise as, or translate to concrete implementations.

The drawbacks of informal specifications are well known — it could be said that 'informal specification' is an oxymoron, and that nothing better than a description can be given informally.

Transition systems have a well defined semantics, but quickly become unwieldy as the complexity of the system being specified increases. Even expressing a simple `JUMP` instruction leads, at best, to an inelegant use of the formalism. Functional languages also have a well defined formal semantics, but are also in general too far from concrete architectures to allow for intuitive specifications. Most current implementations of functional languages are also too inefficient for realistic prototyping. At the other end of the scale, a specification in some imperative language will be easy to implement with reasonable efficiency, but will suffer from a lack of a flexible and useful formal semantics. The same is true, possibly to a lesser extent, of ISPS and register transfer languages. Indeed, despite the existence of an ISP specification [19], early models of the PDP-11 had differing semantics for commands such as the `MOV (R6)+ -(R6)` from example 1, and the semantics of an instruction such as `ADD (R6)+ -(R6)` was not even unambiguously defined.

Most of these methods have been used as a back end for specification systems. The choice of back end is usually between a high level language, with the loss of efficiency and divergence from a final low level implementation that this entails; and a low level language, with the resulting loss of generality and portability. Update Plans are intended as a high level language for specifying low level activities, in such a way that these specifications are relatively simple to transform to specific low level languages. This is achieved by defining a declarative specification language — Update Plans — with an underlying imperative machine model. This makes update plans suitable both for specifying low level activities and as a back end for other specification formalisms. The *archetype* mechanism, which adds a macro-like mechanism to update plans, and which is introduced in chapter seven, possibly weakens the direct link between specification and implementation, but, as illustrated in chapter eight, provides a powerful mechanism for writing specifications which, by means of transformations within the same formalism, lead to easily implementable specifications.

At least two other specification languages combining imperative and declarative features have been developed [12, 13, 27, 29], but in these cases it is the underlying model which is declarative and the surface structure which has a strong imperative flavour. Reversing this, as in Update Plans, leads to a model in which it is easy to reason, and yet in which essentially imperative machine primitives can easily be expressed and combined.
A possible application is an update plan based version of a peephole optimiser [15, 16, 26]. In the peephole optimiser ISP specifications of neighbouring commands are combined and an attempt is made to identify the resulting specification with some other command. The referential transparency of Update Plans is much more suited to such a process than imperative formalisms such as ISP and ISPS. The specification and proof of such an optimiser would make use of the techniques demonstrated in chapter eight.

Update Plans combine the facility of use of imperative languages with the formality of declarative languages, by providing a high level declarative language with an underlying low level imperative model.

3 Some Examples

The two scheme update plan in example 2(a) computes the greatest common divisor of the number initially between `A` and `B` and that initially between `B` and `C`.

Example 2 (a)

```
A[x]B B[y]C =[ x < y ]⇒ B[y − x]C.
A[x]B B[y]C =[ y < x ]⇒ A[x − y]B.
```

Capitalised words denote (unspecified) constants: `A`, `B` and `C` are fixed locators, and `x` and `y` are variables. In fact, if at any stage of the computation we have `A[9]B` and `B[6]C`, (only) the update rule in example 2(b) is applicable, whereupon the `9` is replaced by a `3`.

Example 2 (b)

A[9]B B[6]C⟹A[3]B.

The locators may likewise be specified by variables and obtain their actual value by instantiation, as illustrated by the (nondeterministic) update plan in example 3 which, in the context of a set of appropriate typing rules, sorts the sequence initially between A and C.

Example 3

A[xs]a a[x]b b[y]c c[ys]C =[x > y]⇛ a[y]b b[x]c.

By simple notational conventions "irrelevant" addresses may be omitted and adjacent locator expressions combined (the concept "irrelevant" is more precisely defined in chapter two). Another convention, acknowledging the existence of a *programme counter* at a fixed locator, but hiding it, allows certain update schemes to be written as so-called *commands*. Any command exhibiting the pattern

PC[pc]p pc[*OP args*]qc ... =[...]⇛ PC[pc′] pc′[*next*] ... .

may be written

OP args ... =[...]⇛ *next* ... .

as, for example, in the update schemes in example 4 which may be part of the description of some zero-address machine. (This example anticipates some of the syntactic sugar introduced in chapter two.)

Example 4

PUSH x S[q]⟹S[p] p[x]q.
ADD S[q] [x y]q⟹S[p] p[x + y]q.

These conventions are explained in more detail in chapter two. The command style of writing update schemes will, in chapter seven, be shown to be a special case of the archetype mechanism.

4 Overview

This thesis constitutes a definition of, and tutorial in Update Plans. Chapters two and three formally define basic Update Plans, chapter two covering the syntax and syntactic sugar, and chapter three the semantics. Both of these chapters are illustrated by examples, as are chapters seven and nine in which, respectively, a macro-like mechanism and a degree of synchronous parallelism are added to the formalism.

A longer example, in which an abstract machine for an extended λ-calculus is defined is to be found in chapter four.

A typing mechanism for Update Plans is defined in chapter five. One application of typing is in detecting certain common types of memory use, and this is also covered in chapter five. These paradigms of memory use have well defined semantic properties. Some of these properties are presented in chapter six.

Basic Update Plans, as defined in chapters two and three, are well suited to the specification of abstract machines, but specifications of concrete machines can quickly become unwieldy due to the plethora of combinations of opcodes and addressing modes. The introduction of the archetype mechanism in chapter seven makes it easier to abstract away from this unwanted detail. Chapter eight, in which a register allocation algorithm is shown to preserve semantics, contains larger examples of applications of the archetype mechanism, and of the techniques developed in chapters five and six.

The archetype mechanism creates new possibilities, other than abstracting away from addressing modes, such as specifying asynchronous parallel processors. These possibilities are also covered in chapter seven. Another extension, allowing synchronous parallelism to be specified, is presented in chapter nine and illustrated in a specification of pipelining in the Berkeley RISC II machine.

It is intended that Update Plans be realisable as a full programming language, and as a first step in this direction some implementational aspects are reviewed in chapter ten.

Finally, chapter eleven contains the conclusions, and some suggestions for further research.

5 Notational Conventions

The following notational conventions will be observed throughout this thesis.

- Chapter numbers are always spelt out. All other reference numbers are expressed as numerals. This is, for example, 'section 5 of chapter one'. Sections, figures, examples, etc. are all numbered within chapters. When referring to a section, figure, example, etc. not in the current chapter, the chapter reference will be given explicitly.

- In the running text single quotes are used to indicate strings, double quotes to indicate concepts — i.e. 'string' represents the sequence of symbols 's', 't', 'r', 'i', 'n' and 'g', and is an

example of a "string".

- A typewriter font is used for update schemes. An italic font in an update scheme, e.g. *OP args* in the update schemes presented during the discussion of command style schemes on page 6, indicates a "meta-variable".

- Concatenation of terms to form sequences is usually implicit in Update Plans. When it is necessary to make it explicit the concatenation symbol '++' will be used.

This chapter defines the basic syntax of Update Plans, and introduces some syntactic sugar which makes update plans more readable. An informal description of the semantics of Update Plans was given in chapter one. A formal semantics can be found in chapter three.

1 Basic Syntax

The basic syntax of Update Plans is given by the following grammar.

⟨script⟩ → ⟨configuration⟩ **.** ⟨plan⟩

⟨plan⟩ → ⟨scheme⟩*

⟨configuration⟩ → ⟨locator expression⟩*

⟨scheme⟩ → ⟨configuration⟩ ⟨guard⟩ ⟨configuration⟩ **.**

⟨guard⟩ → =[⟨term⟩]⇒

⟨locator expression⟩ → ⟨locator⟩ [⟨text⟩] ⟨locator⟩

⟨locator⟩ → ⟨term⟩

⟨text⟩ → ⟨term⟩

A term is an expression built from constants, variables and operators, or a regular expression over the set of terms — e.g. 0* represents a sequence of zeros. A regular expression over constants is called a *semi-constant*. If the regular expression contains no closures it is a *finite semi-constant*..

This grammar will be amended and expanded as extensions to Update Plans are introduced. A complete grammar for Update Plans can be found in appendix II.

Whitespace is used to separate the elements of a sequence in the above expressions — i.e. elements in sequences are separated by whitespace,

rather than commas or any other symbol. There must not be any whitespace between a locator and its brace ('[' or ']'), since this would lead to ambiguities. A tie character '~' may be used to override this, so that '...]~ L ...' is equivalent to '...]L ...'.

Rather than addressing the cells as if they were elements in an array, Update Plans address the *boundaries* between cells, so that what in an array notation would be specified as $[2:4] = [\texttt{'a'}, \texttt{'b'}, \texttt{'c'}]$ is specified in an update scheme by `2['a' 'b' 'c']5`. In this locator expression `2` is the *left locator* and `5` the *right locator* of the sequence of cells containing 'a', 'b' and 'c'. However, when no confusion can arise, a cell may be referred to by its left locator. That is, rather than saying *"the cell with left locator X contains the value Y"* one may say *"X contains Y"*. In exceptional cases the right locator may be used in the same way, but then extra care must be taken to avoid confusion.

A singleton sequence is identified with its element. Everything in memory is considered to be a sequence, and singleton sequences are "desequenced" (to a locator, number or other basic type) as the context requires.

A variable is indicated by a lower case identifier. Constants are given by a value or, symbolically, by upper case identifiers. An identifier must start with a letter, and may comprise letters, digits, and the symbols '_' and '′'. Identifiers may be subscripted. Thus `5`, `'s'` and `PC` are all valid constants, `x`, `new_pc` and `i′` are all valid variable names and `Num` and `TeX` are neither. Constant locators play such a special rôle in Update Plans that a special, somewhat suggestive terminology is used to refer to them: a constant locator is called a *register*. By application of the convention that a cell may be referred to by its locator, a cell having a register as left or right locator may also be referred to as a register, if no confusion may arise as to which cell is being referred to.

2 Sweetening the Syntax

An update scheme conforming to the basic syntax can become unwieldy, as shown in example 1(a). While correct, the update schemes in example 1(a) lose legibility due to an excess of detail. Some extensions to the basic syntax will be introduced which provide abstraction capabilities and improve the readability of Update Plans. It is recommended that the reader first consider the simplified versions before spending too much time on the specification in example 1(a).

——— Example 1 (a) ———

The aim of this example is to construct a parser for a context free grammar. The grammar used in the example is

$$\langle \mathrm{S} \rangle \;\rightarrow \mathbf{a} \mid \mathbf{a}\ \langle \mathrm{S} \rangle$$

A script consisting of the following initial configuration

```
PARSE[START]PARSE′ START[S]END END[STOP]END′
    IP[I]IP′ I[ω]EOF
```

where ω is some string, and the three update schemes

```
PARSE[v]PARSE′ v[S]w
    =[TRUE]⇒ PARSE[v′]PARSE′ v′[‘a’]w.

PARSE[v]PARSE′ v[S]w
    =[TRUE]⇒ PARSE[v′]PARSE′ v′[‘a’]v″ v″[S]w.

PARSE[v]PARSE′ v[‘a’]w IP[ip]IP′ ip[‘a’]ip′
    =[TRUE]⇒ PARSE[w]PARSE′ IP[ip′]IP′.
```

will have a final development containing

```
PARSE[END]PARSE′ IP[EOF]IP′
```

if and only if ω is in the language generated by S in the given grammar.

The extensions to the basic syntax fall into two categories. Firstly there is the syntactic sugar — notational changes which increase readability without introducing new capabilities. Secondly there are those notational conventions which also extend the expressive power of Update Plans.

3 Syntactic Sugar

There are four flavours of syntactic sugar, which will be illustrated by their application to the update scheme

Example 1 (b)

```
PARSE[v]PARSE′ v[S]w
    =[TRUE]⇒ PARSE[v′]PARSE′ v′[‘a’]v″ v″[S]w.
```

- Superfluous locators may be omitted. A locator is superfluous if its removal does not lead to any confusion. Using terminology to be introduced in chapter five, the locator's removal may not alter the status with respect to grounding of any other expressions in the update scheme. In the following only right locators have been removed, but left locators may of course, under the same conditions, also be omitted.

Example 1 (c)

```
PARSE[v] v[S]w
   =[TRUE]⇒ PARSE[v′] v′[‘a’]v″ v″[S]w.
```

- Contiguous sequences may be concatenated. Two expressions $x[s]y$ and $y[t]z$ may then be written as $x[s]y[t]z$.

Example 1 (d)

```
PARSE[v] v[S]w
   =[TRUE]⇒ PARSE[v′] v′[‘a’]v″[S]w.
```

- Locators may also be omitted when concatenating contiguous sequences so that $x[s]y[t]z$ may also be written as $x[s\ t]z$, again if no confusion may arise.

Example 1 (e)

```
PARSE[v] v[S]w
  =[TRUE]⇒ PARSE[v'] v'['a' S]w.
```

- Tautological guards may be omitted.

Example 1 (f)

```
PARSE[v] v[S]w ⟹ PARSE[v'] v'['a' S]w.
```

- Identical left hand sides may be shared. A *repeat* of the previous left hand side is indicated by the "repeat" symbol '"'.

Example 1 (g)

```
PARSE[v] v[S]w  ⟹ PARSE[v'] v'['a']w.
       "        ⟹ PARSE[v'] v'['a' S]w.
```

4 Notational Conventions

There are three, more far reaching, notational conventions. These are used for specifying input and output activities, for increasing the readability of a common type of update scheme called a *command*, and to allow more compact specifications.

I/O. The update scheme for parsing the terminal symbol 'a' included a mechanism for reading input, by means of an input pointer.

Example 1 (h)

```
PARSE[v] v['a']v' IP[ip] ip['a']ip'
  ⟹ PARSE[v'] IP[ip'].
```

The scheme contains a register IP which points to an input stream. When 'a' is read the pointer is moved over that symbol. The symbol '?IP' is used as shorthand for this mechanism. Any update scheme exhibiting the pattern

$$\ldots\ IP[i]\ \ i[input]j\ \ \ldots\ \Longrightarrow\ \ldots\ \ IP[j]\ \ \ldots\ .$$

may then be written

$$\ldots\ ?IP[input]\ \ \ldots\ \Longrightarrow\ \ldots\ .$$

Though this convention could also be used for standard stack operations, its use is restricted to operations that conceptually can be considered to be input operations. Chapter 7 specifies a macro-like mechanism in which this and many other types of addressing structure can be defined.

A further refinement is that `?[`*input*`]` can be used for the standard input stream, when this is unambiguously defined.

Note that if the input pointer is not moved — the input is looked at, but not read — then this convention cannot be used.

There is a similar convention for output, using the symbol '`!`'. Any scheme with the pattern

$$\ldots\ OP[o]\ \ldots\ \Longrightarrow\ \ldots\ o[output]p\ \ OP[p]\ \ldots\ .$$

may be written as

$$\ldots\ \Longrightarrow\ \ldots\ \ !OP[output]\ \ \ldots\ .$$

Again `![`*output*`]` may be used for the standard output stream, if this is defined. The scheme in example 1(h) can now be written:

Example 1 (i)

```
PARSE[v] v['a']w ?['a'] ⟹ PARSE[w].
```

Programme counter. The update schemes obtained by applying all the conventions above satisfy the requirements, namely specifying a parser for the given grammar, but the introduction of new registers, such as `PARSE`, for each routine one wants to implement leads to illegibility and makes re-use of specifications in disparate contexts difficult. This can be simplified by introducing a command stream and a programme counter to administer it.

Example 1 (j)

The schemes above are summarised here for convenience.

```
PARSE[v] v[S]w  ⟹ PARSE[v'] v'['a']w.
      "         ⟹ PARSE[v'] v'['a' S]w.
PARSE[v] v['a']w ?['a'] ⟹ PARSE[w].
```

These can be transformed to a command stream style by replacing the constant locator `PARSE` by the constant "command" `PARSE` giving

```
PC[pc] pc[PARSE v]qc v[S]w
    ⟹ PC[pc'] pc'[PARSE v']qc v'['a']w.
PC[pc] pc[PARSE v]qc v[S]w
    ⟹ PC[pc'] pc'[PARSE v']qc v'['a' S]w.
PC[pc] pc[PARSE v]qc v['a']w ?['a']
    ⟹ PC[pc'] pc'[PARSE w]qc.
```

The initial configuration becomes

```
PC[GO] GO[PARSE START]
    START[S]END END[STOP] IP[I] I[ω]EOF
```

The string ω is in the language generated by S if there is a final development satisfying

```
PC[pc] pc[PARSE END] IP[EOF].
```

At first sight introducing a programme counter does not seem to have increased the legibility of the update schemes, but the following notational convention, already introduced in chapter one, puts that right.

If an update scheme exhibits the pattern

PC[pc] pc[*command*]qc ...
    =[...]⇛ PC[pc′] pc′[*next*]qc ....

where *command* and *next* are some, possibly empty, sequences, and the locators PC, pc, qc and pc′ can be omitted without confusion, it may be rewritten as

OP args ... ⟹ *next* ....

An update scheme of this type is called a *command*. The first term of *command* will usually be some constant, known as the *opcode*. The sequences are known as *command sequences*. An update plan in which all update schemes are commands is called a *command driven* plan. The convention may also be applied individually to the left or right hand side, if the other side is not in command form. A configuration is in *command form* if it contains a non-empty command sequence, or if the contents of PC (the contents of the cell with left locator PC) are not specified. When only one side of an update scheme need be desugared the locator qc is not shared.

In most cases it will be necessary to apply this convention to the left and right hand sides simultaneously in order to satisfy the conditions under which locators may be omitted. The PARSE update scheme under consideration is a case in point. Rewriting only the left hand side gives

```
PARSE v v[S]w ⟹ PC[pc′] pc′[PARSE v′]qc v′[‘a’ S]w.
```

which contains the undefined (or non-ground, see chapter five) locators pc′ and qc. Similarly, rewriting only the right hand side gives

```
PC[pc] pc[PARSE v]qc v[S]w ⟹ PARSE v′ v′[‘a’ S]w.
```

which, when desugared to

```
PC[pc] pc[PARSE v]qc v[S]w
    ⟹ PC[pc′] pc′[PARSE v′]qc′ v′[‘a’ S]w.
```

contains the non-ground locators pc′ and qc′. Simultaneously rewriting both sides gives the update plan

Example 1 (k)

```
PARSE v v[S]w  ⟹  PARSE v′ v′[‘a’]w.
      "        ⟹  PARSE v′ v′[‘a’ S]w.

PARSE v v[‘a’]w ?[‘a’] ⟹PARSE w.
```

Detecting command style update schemes is (almost) a trivial syntactic exercise. If a non-empty command is present it must be desugared as described above. An empty command is assumed to be present if the contents of `PC` are not specified. Note that the contents of `PC` may also be specified when a non-empty command is present, making the contents of `PC` available in a command style update scheme. When both the left and right hand side are desugared the right locators of both commands must be identical. The following examples should make the desugaring mechanism clear.

Example 2

The update scheme

```
JUMP l ⟹ PC[l].
```

becomes

```
PC[pc] pc[JUMP l]qc ⟹ PC[l].
```

The left hand side is desugared, the right hand side isn't, since there is no non-empty command, and the contents of `PC` are specified. The update scheme:

```
CALL l PC[p] SP[t] ⟹ PC[l] SP[s] s[p]t.
```

becomes

```
PC[pc] pc[CALL l]qc PC[p] SP[t]
       ⟹ PC[l] SP[s] s[p]t.
```

Again the left hand side is desugared and the right hand side isn't. Note that consistency of the instantiation of the left hand side insures that `p` is equal to `pc`.

```
⟹ SILLY.
```

becomes

```
PC[qc] ⟹ PC[pc] pc[SILLY]qc.
```

Both sides are desugared. The left hand side contains the empty command.

Alternatives. An update plan may contain update schemes implementing some form of case analysis — a set of update schemes having part or all of their left hand sides in common, and distinguished from one another by mutually exclusive implicit or explicit guards. This may often entail repetitive occurrences of these guards, as in

$$
\begin{array}{lll}
lhs_1 & =[g_1]\Rightarrow & rhs_1\,. \\
lhs_2 & =[\neg g_1 \wedge g_2]\Rightarrow & rhs_2\,. \\
 & \vdots & \\
lhs_n & =[\neg g_1 \wedge \neg g_2 \wedge \ldots \wedge g_n]\Rightarrow & rhs_n\,.
\end{array}
$$

Such schemes may be written as *alternatives*, in which only the first applicable update scheme, reading from top to bottom, will be applied. Alternatives are separated by semi-colons. The schemes above can be written

$$\begin{array}{lll} lhs_1 & =[g_1]\Rightarrow & rhs_1\,; \\ lhs_2 & =[g_2]\Rightarrow & rhs_2\,; \\ & \vdots & \\ lhs_n & =[g_n]\Rightarrow & rhs_n\,. \end{array}$$

Alternatives can be eliminated from an update plan by replacing the guards that have been omitted. Implicit guards will be detected by the typing mechanism (see chapter five).

5 Further Simplifications

As an aside, note that the update schemes in example 1(k) can be simplified further, at the cost of a simple change to the initial configuration. The first step is to eliminate the indirection in `PARSE`'s argument, which gives the update script shown in example 1(l).

Example 1 (l)

Elimination of indirection gives the update plan

```
PARSE S  ⟹ PARSE 'a'.
   "     ⟹ PARSE 'a' S.

PARSE 'a' ?['a'] ⟹ PARSE.
```

The end of the parse now has to be indicated in the initial configuration by modifying this to

```
PC[START] START[PARSE S STOP]
  IP[I] I[ω]EOF.
```

A final development must now satisfy

```
PC[pc] pc[PARSE STOP] IP[EOF].
```

The `PARSE` command has now become superfluous since both `S` and `'a'` are constants and can therefore both be used as opcodes.

Example 1 (m)

```
S  ⟹ 'a'.
"  ⟹ 'a' S.

'a' ?['a'] ⟹ .
```

The initial configuration is

```
PC[START] START[S STOP] IP[I] I[ω]EOF.
```

A final development should satisfy

```
PC[pc] pc[STOP] IP[EOF].
```

6 Summary

The syntax of Update Plans, with syntactic sugar as defined in this chapter, is given by the following context free grammar. This grammar makes use of the notational conventions of the specification formalism ASF+SDF [36, 37] in which the notation $\{S\ term\}^+$ indicates a list of one or more S's separated by the terminal $term$. If the $^+$ is replaced by a * the list may also be empty. The suffix *opt* indicates zero or one occurrences of its nonterminal.

⟨script⟩ → ⟨configuration⟩ **.** ⟨plan⟩

⟨plan⟩ → ⟨item⟩*

⟨item⟩ → ⟨alternatives⟩ **.**

⟨alternatives⟩ → {⟨scheme⟩ **;**}$^+$ **.**

⟨scheme⟩ → ⟨configuration⟩ ⟨guard⟩ ⟨configuration⟩ |
⟨repeat⟩ ⟨guard⟩ ⟨configuration⟩

⟨configuration⟩ → ⟨text⟩ ⟨context⟩

⟨text⟩ → ⟨term⟩*

⟨context⟩ → ⟨locator expression⟩*

⟨repeat⟩ → "

⟨guard⟩ → =[⟨term⟩]⇛ | ⟹

⟨locator expression⟩ →
⟨left-section⟩$^+$ ⟨locator⟩ |
? ⟨term⟩-*opt* [⟨contents⟩] |
! ⟨term⟩-*opt* [⟨contents⟩]

⟨left-section⟩ → ⟨locator⟩ [⟨contents⟩]

⟨locator⟩ → ⟨term⟩-*opt*

The basis for Update Plans is a formal notation — locator expressions — for specifying machine configurations. These are then combined in update schemes, which specify possible machine transitions in which the machine is updated from a configuration specified by the left hand side of the update scheme to a configuration specified by the right hand side. Chapter two defined the syntax of Update Plans. In this chapter the semantics of Update Plans is formally defined.

This chapter consists of two sections. Section 1 discusses representations of computer memory and relates locator expressions to machine configurations. In section 2 this relation is extended to cover the full syntax of Update Plans.

1 Memory

Computer memory can be viewed in two complementary ways. In the first view, the machine on which update plans operate consists simply of a *memory* containing a finite number of stores, each consisting of a countable set of *cells* addressed by a completely ordered set of *locators*. Since these multiple stores can easily be mapped to a single store the machine will in the following be considered to consist of only one store.

In the second view, the memory is a function $\mathcal{M}$ from locators to values. The domain of $\mathcal{M}$, or address space $\mathcal{A}$, is a countable set of locators. Since it is a countable set, there is a complete ordering $<_{\mathcal{A}}$ along with the relevant successor and predecessor functions $\mathsf{succ}_{\mathcal{A}}$ and $\mathsf{pred}_{\mathcal{A}}$. In update plans '+' and '−' are used in the usual way: $L + n$ to indicate $\mathsf{succ}^n_{\mathcal{A}}\ L$ and $L - n$ for $\mathsf{pred}^n_{\mathcal{A}}\ L$. The address space is two way infinite, that is $\mathsf{succ}_{\mathcal{A}}$ and $\mathsf{pred}_{\mathcal{A}}$ are defined for all elements of $\mathcal{A}$.

1.1 Locators

In the following the term 'memory' will be used to refer to the whole memory of the machine. This means that, in the formal view, $\mathcal{M}$ is a total function, possibly returning some special value $\perp$ for undefined values. A finite subset of the memory function will be referred to as a 'configuration'.

A configuration is *satisfied* by any memory or configuration of which it is a subset, and only by such a configuration or memory. The domain of a configuration c — the subset of $\mathcal{A}$ for which c is defined — is given by $\mathsf{Dom}\ c$. Update schemes define relations between configurations. An application of an update scheme to a memory defines a relation between memories.

A locator expression defines a configuration. The locator expression `L[S]L'` expresses the idea *"the cells between L and L' contain the sequence S"*. Since the locator expression contains no information about the cells other than those between L and L', only the values in the cells between L and L' are defined. More formally, there is a function I from locator expressions to configurations given by

$$I[\![l[s_0 \ldots s_n]r]\!] = \{(l+i, s_i) \mid 0 \leq i \leq n \wedge l + n = r\}.$$

Locator expressions can be combined to specify larger configurations. A set of locator expressions only specifies a configuration if it is *consistent*, i.e. there are no locator expressions in the set specifying conflicting contents for one and the same cell. For example `3[1 2]5 4[2 3]6` is consistent, but `3[1 2]5 4[1 2]6` is not, due to the conflict between `4[2]5` and `4[1]5`.

Consistency can be defined formally as follows. Two configurations c and c' are consistent if $c\ x = c'\ x$ for all x in $(\mathsf{Dom}\ c) \cap (\mathsf{Dom}\ c')$. A set of locator expressions L is consistent if $I[\![l]\!]$ is consistent with $I[\![l']\!]$ for all l and l' in L.

1.2 Combining Configurations

Configurations can be combined in a manner akin to set union. The "update union" of two configurations must also be a configuration, i.e. a partial function. If the two configurations to be combined are not consistent then, by definition, there is an argument for which they give different values. One of the configurations is chosen to consistently provide the result in such cases. By convention this is the left argument of the "update union" operator.

Formally

$$c \uplus c' = c \cup (c' \upharpoonright (\mathsf{Dom}\ c' \setminus \mathsf{Dom}\ c)),$$

where $f \upharpoonright S$ is the restriction of the function f to the set S. This definition is equivalent to

$$\begin{aligned}(c \uplus c')\ l &= c\ l, \text{ if } l \in \mathsf{Dom}\ c \\ &= c'\ l, \text{ if } l \notin \mathsf{Dom}\ c\end{aligned}$$

Note that $\uplus$ is equivalent to conventional set union if the configurations to which it is applied are mutually consistent.

The union operator defined above is used to extend the function $I[\![\cdot]\!]$ to cover collections of locator expressions. This extension is noted $\mathcal{I}[\![\cdot]\!]$, and is defined by

$$\mathcal{I}[\![e_1 \ldots e_n]\!] = I[\![e_1]\!] \uplus I[\![e_2]\!] \uplus \ldots \uplus I[\![e_{n-1}]\!] \uplus I[\![e_n]\!]$$

The motivation for this definition can be found in section 2.

2 Update Schemes

Locator expressions are the basic building blocks of update schemes. The locator expressions presented above contain only constants, or instantiated values such as `3` and `'b'`. This is not the case in update schemes. These may contain variables, or uninstantiated values.

A locator expression containing variables is instantiated by substituting values for the variables to obtain a closed locator expression. A *substitution* is a mapping from variables to values. A value is a closed term. Given an expression e and a substitution σ, the instantiation of e by σ is noted e^{σ}.

An update scheme is constructed from two sets of locator expressions, forming the left and right hand sides, and a guard.

Example 1

Consider the following update scheme:

```
PC[x] x[BCC o]y C[0] ⟹PC[y+o].
```

This is a definition for the `BCC` instruction on the M6800 processor. Intuitively the definition states that if the instruction addressed by the `PC` is `BCC` with argument `o` and the condition code register `C` bit is `0` then the `PC` is reset to `y + o` where `y` is the address immediately after the `BCC o` instruction.

More formally, this update scheme can be applied to the memory $\mathcal{M}$ if the following conditions hold.

$$\begin{aligned} \mathcal{M}\ x &= \texttt{BCC}, \text{ where } x = \mathcal{M}\ \texttt{PC} \\ \mathcal{M}\ c &= \texttt{0}, \text{ where } c = \mathcal{M}\ \texttt{C} \end{aligned}$$

or equivalently $((\mathcal{M} \circ \mathcal{M})\ \texttt{PC} = \texttt{BCC}) \wedge (\mathcal{M}\ \texttt{C} = \texttt{0})$. If these conditions hold then the result of applying the update scheme is a new memory $\mathcal{M}'$ defined as follows.

$$\begin{aligned} \mathcal{M}'\ \texttt{PC} &= y + o \\ &\quad \text{where } y = x + 2 \\ &\qquad\quad o = \mathcal{M}\ (x+1) \\ &\qquad\quad x = \mathcal{M}\ \texttt{PC} \\ \mathcal{M}'\ x &= \mathcal{M}\ x, \text{if } x \neq \texttt{PC} \end{aligned}$$

An update scheme is *applicable* to a configuration if there is a substitution under which the guard is true, and the left hand side of the update scheme is contained in the configuration. The result of applying the update scheme is then this configuration, minimally changed so as to contain the instantiation of the right hand side, under the same substitution as that applied to the left hand side and the guard. The precise conditions under which such a substitution can be found will be discussed in chapter five, in which *grounding* is introduced, but, intuitively, a variable is ground if a closed expression can be derived for it. In fact a less restrictive requirement, *semi-grounding*, is defined in chapter five, but for this example the concept of grounding is sufficient.

The following example should clarify these concepts. Consider the update scheme for the `BCC` instruction of example 1, and the configuration `PC[1234] 1234[BCC 67]1236 C[0]`. A substitution, $\{(x, 1234), (y, 1236), (o, 67)\}$, can be found under which the left hand side is contained in, and in fact is identical to the given configuration, and the update scheme is therefore applicable to the configuration. The result is then the configuration `PC[1303] 1234[BCC 67]1236 C[0]`. Only the contents of `PC` have been changed, this being the minimal change necessary to make the configuration satisfy the right hand side of the update scheme under the substitution above. The update scheme is also well formed. All variables occurring in it are ground. An expression for o, for example, is $\mathcal{M}\ ((\mathcal{M}\ \texttt{PC}) + 1)$.

More formally, an update scheme $(lhs, guard, rhs)$ is applicable to a configuration c if there is a substitution σ such that rhs^σ is consistent, $lhs^\sigma \subseteq c$ and $guard^\sigma$ holds. The result is the configuration c' given by

$$c' = \mathcal{I}[\![rhs^\sigma]\!] \uplus c.$$

In this configuration one can distinguish three types of cells. There are those cells specified in the configuration c, but not in the right hand side of the update scheme. The values in these cells remain unchanged. Secondly, there are the cells specified both in c and the right hand side of the scheme. The values in these cells are *updated* to the values specified in the right hand side, though they need not, of course, actually change. Finally, there may be some cells specified in the right hand side of the update scheme that do not occur in c. These cells are then added to the configuration.

The motivation for the definition of $\uplus$, and its application in the definition of $\mathcal{I}[\![\cdot]\!]$, should now be clear. The left argument of $\uplus$ specifies an update of the right argument. Operationally, any cells in memory specified in the left configuration must have their value updated to the value specified in that configuration. The values in cells not specified in the left argument are left unchanged.

A substitution must now be found. The substitution must satisfy the guard and, when applied to the left hand side, it must give a configuration contained in the configuration to which the update scheme is being applied. More formally, a set of locator expressions L, a guard g and a configuration c together define a set of applicable substitutions as given by the function $\mathcal{S}$.

$$\mathcal{S}\ L\ g\ c = \{\sigma \mid (\mathcal{I}[\![L^\sigma]\!] \subseteq c) \wedge g^\sigma\}$$

Note that consistency of L under the substitution is not explicitly required. However, if c is consistent the instantiation of L must, implicitly, also be consistent.

The concept of applying update schemes can now be formalised. An update scheme $s = (lhs, guard, rhs)$ defines a relation between configurations, notation $\Rightarrow_s$, given by

$$c \Rightarrow_s c' \equiv \exists \sigma \in (\mathcal{S}\ lhs\ guard\ c) : rhs^\sigma \text{ is consistent} \wedge c' = \mathcal{I}[\![rhs^\sigma]\!] \uplus c.$$

When no ambiguity can arise the subscript s may be omitted. The relation $\Rightarrow_s$ defines all configurations c' that can be derived from configuration c by application of update scheme s. There must be a substitution, delivered by the function $\mathcal{S}$, under which the left hand side of s is a subset of c, and under which the guard of s is true. The result of applying s is then c, minimally updated to be consistent with the instantiation, under the same substitution, of the right hand side of s.

One update scheme is typically used to specify one possible machine transition, for example, the effect of one specific instruction. A complete machine specification, such as that in chapter four, will contain many such update schemes. The transition relation defined above can easily be extended to update plans. Any update scheme, contained in the update plan, which is applicable to a given configuration may be applied to that configuration. An update plan P defines the following relation.

$$c \Rightarrow_P c' \equiv \exists s \in P : c \Rightarrow_s c'.$$

In $\Rightarrow_P$ a nondeterministic choice is made of which update scheme to apply. Again, the subscript may be omitted when no confusion can arise. As usual $\Rightarrow^*$ is the reflexive transitive closure of $\Rightarrow$.

A configuration c' is a *development* of configuration c under an update plan P if $c \Rightarrow_P^* c'$. A *final development* of a script consisting of initial configuration c and update plan P is any configuration c' such that $c \Rightarrow_P^* c'$ and $\neg\exists c'' : c' \Rightarrow_P c''$. To arrive at a final development one keeps on applying update schemes until one arrives at a configuration to which no update schemes in the plan are applicable — i.e. a final development is analogous to a normal form in a term rewrite system.

Chapter Four An Introductory Application and Example

In this chapter a slightly more complex example than those already presented is given. In this example an abstract machine, based on the λMMachine [52], is specified in terms of an update plan. The λMMachine is an abstract machine for the λ-calculus. It is a register based graph reduction machine using environments. The machine has been extended to handle constants, strict abstraction, some predefined functions and simple i/o. This extended λ-calculus is called the λ^+-calculus. The λ^+-calculus machine, or λ^+MMachine, is provided with some syntactic sugar, providing global and local function declarations and infix operators. The result is a simple functional language called 'FLIP', for 'Functional Language Implementation Prototype'. The acronym can also be seen as describing the four stages of compilation, as shown in figures 1 and 2:

- **F**unctions (or **F**LIP itself)
- **L**ambda calculus
- **I**ntermediate code
- **P**rogramme (or update **P**lan)

The λMMachine was chosen as an example for several reasons.

- Functional languages have a simple syntax — a complex syntax would unnecessarily complicate the presentation.
- Functional languages present non-trivial implementation problems.
- Update Plans are intended for specifying those implementational details, primarily of a memory management nature, which would only obfuscate the specification given by Meijer and Patterson [52].

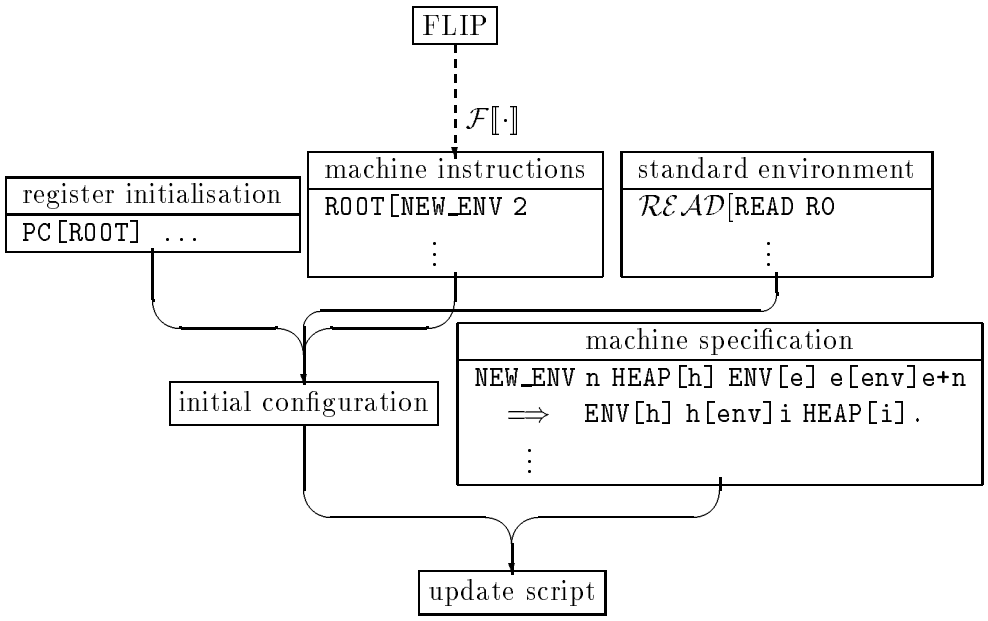


Figure 1: Compiling FLIP, Code Sequence to Update Script

The following section discusses compiling λMMachine programmes to FLIP machine code, and gives an overview of the remainder of this chapter.

1 Compiling λMMachine Programmes to FLIP code

A FLIP programme is compiled to a configuration — a specification of the contents of the programme store — for the FLIP machine. This is then combined with programme independent initialisations — register initialisations (given on page 28) and a standard environment — to form an initial configuration. This initial configuration, in combination with the update schemes specifying the instruction set of the FLIP machine (summarised in appendix III), forms an update script, execution of which will terminate in a configuration containing a representation of the value of the original FLIP programme. This is discussed in more detail below, and represented schematically in figure 1.

In figure 1 $\mathcal{F}$ is the compilation function. $\mathcal{F}$ is the composition of three auxiliary functions, $\mathcal{D}$, $\mathcal{E}$ and $\mathcal{T}$, with $\mathcal{F}[\![f]\!] = \mathcal{T}\,[\![\mathcal{E}[\![\mathcal{D}[\![f]\!]]\!]\ \sigma\ \tau]\!]$ `ROOT`, where σ and τ are lists of register names, τ being an infinite list of free registers, and σ a list of registers that have been pushed to a compile time stack. `ROOT` is the locator in `PC` in the initial configuration (see page 28). $\mathcal{D}$ is not given explicitly. It is just a desugaring of FLIP syntax. $\mathcal{E}$ translates a FLIP programme to intermediate code, and $\mathcal{T}$ compiles this to FLIP machine code. A concrete example of this compilation path is given in figure 2. The example programme in figure 2 includes an application of strict abstraction, indicated by exclamation marks, and a predefined i/o function (`read`). These extensions to the λ-calculus will be covered in sections 4 and 6, respectively. The translation of the intermediate command $STORE\ x$ to the code sequence `POP R0 BIND 0 R0` is discussed

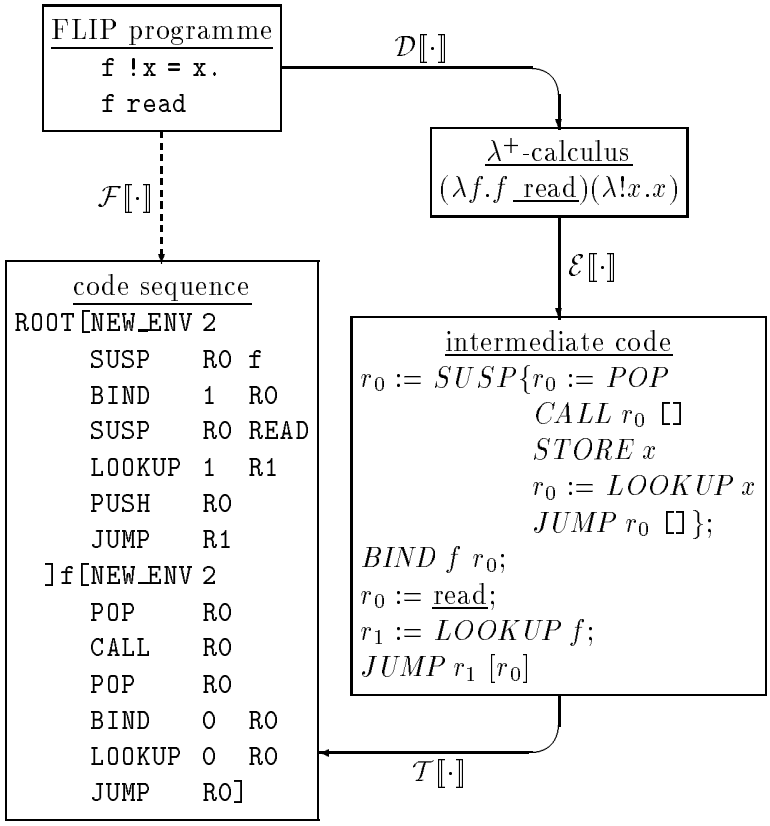


Figure 2: Compiling FLIP, FLIP to Code Sequence

in section 4.4.

$\mathcal{F}$ is specified incrementally. Section 2 gives $\mathcal{F}$ for the kernel of the language, section 3 defines the extension of $\mathcal{F}$ for integers, section 4 that for strictness, section 5 for predefined functions, and section 6 completes the specification by defining $\mathcal{F}$ for i/o.

Each section is structured in accordance with the compilation path, and will therefore contain, in the following order,

- a description of the (extension of the) syntax of the λ-calculus, which is the desugared version of (the extension to) FLIP.
- an informal description of the intermediate code produced by $\mathcal{E}$
- the function $\mathcal{E}$ (for this extension)
- the translation function $\mathcal{T}[\![\cdot]\!]$ from intermediate code to abstract machine instructions (again for this extension)
- update scheme specifications of the abstract machine commands which result from $\mathcal{T}$.

The update schemes will be numbered for ease of reference. Due to the incremental nature of the presentation it will sometimes be necessary to replace one update scheme with another. In this case the replacement will

not receive a new number, but will inherit the number of the scheme it replaces.

For any well formed FLIP programme f the code sequence $\mathcal{F}[\![f]\!]$ forms part of the initial configuration of the desired update script. The remainder of the configuration, which is independent of the programme being compiled, consists of a standard environment, discussed in sections 5 and 6, and the following programme independent initialisations.

```
PC[ROOT]
HEAP[H] ENV[H] SP[S] S[END] END[STOP] TP[T] T[S].
```

The significance of the registers `HEAP`, `ENV`, `SP` and `TP` will be made clear during the specification of the machine. The specifications of the abstract machine commands, presented below and summarised in appendix III, form the update plan part of the script. The formation of the update script is shown schematically in figure 1. The script will be unambiguous and, applied if necessary to the input stream, its final development will be a representation of the value of the FLIP programme. Clearly, the value of the programme in figure 2, given input '`2`', is 2. In the following specification a basic value, such as an integer, is represented by a *constructor* somewhere in the store. A constructor is a pair, consisting of the constant `CONST` and the basic value. A programme terminating with such a value will halt at a jump to a constructor containing the value (the reason for this construction can be found in the specification, and has to do with the strict abstraction mechanism). Given input '`2`', the script containing the code in figure 2 will evaluate to a final development satisfying

```
PC[pc] pc[JUMP r] r[h] h[CONST 2].
```

2 The Basic λ-Calculus Machine

2.1 Syntax

The syntax of the λ-calculus is not very relevant to the example in hand, and assumed known.

2.2 Intermediate Code

The λMMachine contains a heap, a stack, an unspecified number of registers and a static code space. A function application is encoded as a *suspension*, consisting of a pointer to a code sequence and a pointer to an environment, in which free variables are bound to their values.

The following intermediate code instructions are needed in order to implement this:

$r := LOOKUP\ x$
: Look up the value of x (the suspension to which x is bound) in the current environment and assign this to register r.

$r := SUSP\ \phi$
: Create a suspension consisting of a pointer to the code ϕ and a pointer to the current environment, and assign it to r.

$r := POP$
: Pop the topmost suspension from the stack and assign it to r.

$BIND\ x\ r$
: Bind x in the current environment to the value of register r.

$JUMP\ r\ [r_1 \ldots r_n]$
: Ensure that the values of registers r_1 to r_n are on the stack, then jump to the code of the suspension bound to r while installing its environment.

2.3 $\mathcal{E}[\![\cdot]\!]$, Translation to Intermediate Code

The translation scheme presented below is adapted from work done by Meijer and Patterson [52], as are the descriptions of the intermediate code.

Arguments of functions are conceptually pushed onto the stack, to be popped when needed. In order to improve efficiency these arguments are, whenever possible, stored in registers rather than being transferred via the stack. In order to do this a compile time "stack" of registers, emulating a run time stack, is maintained. Only when a function is called must the contents of the registers on the compile time stack be pushed onto the run time stack.

The central translation scheme is that for expressions: $\mathcal{E}[\![e]\!]\ \sigma\ \tau$. The translation schemes for definitions, expressions and suspensions are:

$$
\begin{array}{rcl}
\mathcal{E}[\![e_1\ e_2]\!]\ (r +\!\!+ \sigma)\ \tau & = & r := \mathcal{C}[\![e_2]\!]; \\
 & & \mathcal{E}[\![e_1]\!]\ \sigma\ (r +\!\!+ \tau) \\
\mathcal{E}[\![\lambda x.e]\!]\ \sigma\ (r +\!\!+ \tau) & = & BIND\ x\ r; \\
 & & \mathcal{E}[\![e]\!]\ (r +\!\!+ \sigma)\ \tau \\
\mathcal{E}[\![\lambda x.e]\!]\ (r +\!\!+ \sigma)\ [\,] & = & r := POP; \\
 & & BIND\ x\ r; \\
 & & \mathcal{E}[\![e]\!]\ (r +\!\!+ \sigma)\ [\,] \\
\mathcal{E}[\![e]\!]\ (r +\!\!+ \sigma)\ \tau & = & \mathcal{C}[\![e]\!]: \\
 & & JUMP\ r\ \tau
\end{array}
$$

where the auxiliary function $\mathcal{C}$ is used to produce intermediate code for the arguments of an application. $\mathcal{C}$ is:

$$
\begin{array}{rcl}
\mathcal{C}[\![x]\!] & = & LOOKUP\ x \\
\mathcal{C}[\![e]\!] & = & SUSP\ \mathcal{E}[\![e]\!]\ \Sigma\ [\,]
\end{array}
$$

where Σ is the list of all available registers.

See figure 2 for an example of an application of these and other translation schemes.

2.4 $\mathcal{T}[\![\cdot]\!]$, Translation to Machine Configurations

The translation to machine configurations is not difficult. The function $\mathcal{U}$ translates intermediate code to initial configurations. It returns a pair. The first element of the pair is a translation of the instructions occurring in the argument of $\mathcal{U}$; the second is a translation of the suspensions. The function $\mathcal{T}$ ensures that these are concatenated into a correct configuration.

The prototype presented here implements environments in a very simple minded way — each suspension in which new bindings are made creates its own copy of the universal environment, containing fields for all variables in the script. A more sophisticated implementation would probably use a combination of de Bruijn numbering [17] and some dereferencing mechanism. $\mathcal{T}$ is:

$$\mathcal{T}[\![code]\!]\, loc = loc[\texttt{NEW\_ENV}\ \Sigma\ instrs]\ susps$$

where Σ is the size of the universal environment, *instrs* is a code sequence for the function encoded in *code*, and *susps* is a set of locator expressions containing the code sequences of functions called in *code* — i.e. the code sequences of suspensions occurring in *code*. These two translations, *instrs* and *susps* are derived as follows. The function $\mathcal{U}$, defined below, when applied to an intermediate code command, returns a pair consisting of a FLIP machine code sequence implementing this intermediate code command and, for a *SUSP susp r* command, a set of locator expressions implementing the suspension *susp*. When mapped across the intermediate code sequence *code* the function $\mathcal{U}$ will therefore return a list of pairs of FLIP machine code sequences and lists of locator expressions. The pair is brought outside the list — i.e. the list of pairs of lists becomes a pair of lists of lists, and these lists of lists are flattened to give a pair of lists. The first element of the pair is then *instrs* and the second *susps*. A full definition of $\mathcal{T}$ is:

$$\begin{array}{rcl}
\mathcal{T}[\![code]\!]\, loc & = & loc[\texttt{NEW_ENV}\ \Sigma\ instrs]\ susps \\
\quad \text{where}\ \ (instrs, susps) & = & (flatten\ instrs', flatten\ susps') \\
(instrs', susps') & = & lift\ (map\ \mathcal{U}\ code) \\
lift\ prs & = & (map\ fst\ prs, map\ snd\ prs) \\
fst\ (a,b) & = & a \\
snd\ (a,b) & = & b
\end{array}$$

$\mathcal{U}$ is defined for the intermediate code instructions in the translation scheme for $\mathcal{E}$ above as follows:

$$\begin{array}{rcll}
\mathcal{U}[\![r := LOOKUP\ x]\!] & = & (\texttt{LOOKUP}\ \sigma(x)\ \texttt{r}, & \epsilon) \\
\quad \text{where } \sigma \text{ is the indexing function for identifiers in the environment.} & & & \\
\mathcal{U}[\![r := SUSP\ \{susp\}]\!] & = & (\texttt{SUSP}\ \texttt{r}\ l, & \mathcal{T}[\![susp]\!]\ l) \\
\quad \text{where } l \text{ is some unique locator} & & & \\
\mathcal{U}[\![r := POP]\!] & = & (\texttt{POP}\ \texttt{r}, & \epsilon) \\
\mathcal{U}[\![BIND\ x\ r]\!] & = & (\texttt{BIND}\ \sigma(x)\ \texttt{r}, & \epsilon) \\
\mathcal{U}[\![JUMP\ r\ [r_1..r_n]]\!] & = & (\texttt{PUSH}\ \texttt{r}_n & \\
 & & \vdots & \\
 & & \texttt{PUSH}\ \texttt{r}_1 & \\
 & & \texttt{JUMP}\ \texttt{r}, & \epsilon)
\end{array}$$

Again, an example of an application of $\mathcal{T}[\![\cdot]\!]$ can be found in figure 2.

2.5 Machine Instructions

The environmental commands are:

```
[2.1] NEW_ENV n HEAP[h] ENV[e] e[env]e+n
        ⟹ ENV[h] h[env]i HEAP[i].
[2.2] BIND i r r[h] ENV[e] ⟹ e+i[h].
[2.3] LOOKUP i r ENV[e] e+i[h] ⟹ r[h].
```

The stack commands are straightforward.

```
[2.4] PUSH r r[h] SP[t] ⟹ SP[s] s[h]t.
[2.5] POP r SP[s] s[h]t ⟹ r[h] SP[t].
```

The last two commands are `SUSP`, which creates a suspension on the heap, and `JUMP` which jumps to such a suspension.

```
[2.6] SUSP r c ENV[e] HEAP[h]
        ⟹ r[h] h[c e]i HEAP[i].
[2.7] JUMP r r[h] h[c e]
        ⟹ PC[c] ENV[e].
```

3 Basic Values

Only values of the type *integer* will be considered here. The construction used is such that it is fairly simple to extend it to include other types.

3.1 Syntax

Integers are represented by their standard decimal notation.

3.2 Intermediate Code

There is obviously no need to look up the value of a constant in the environment, so a new intermediate code instruction is needed which simply assigns the value of the constant to a register.

$r := c$
: Assign the constant c to register r.

3.3 $\mathcal{E}[\![\cdot]\!]$

A constant is its own translation.

$$\mathcal{C}[\![c]\!] \;=\; c$$

3.4 $\mathcal{T}[\![\cdot]\!]$

Constants can also be represented by suspensions. They will be tagged to distinguish them from ordinary suspensions. A new command is therefore introduced, analogous to the `SUSP` command, to create tagged suspensions containing values.

$$\mathcal{U}[\![r := c]\!] \;=\; (\texttt{ASSIGN r } \mu(c) \quad \texttt{CONST r}, \qquad \epsilon)$$

where μ is a compile time function returning the internal representation of its argument.

3.5 Machine Instructions

The assignment command is straightforward.

[3.1] `ASSIGN r n ⟹ r[n].`

The suspensions created by the `CONST` command are tagged to indicate that they are constructor suspensions.

[3.2] `CONST r r[n] HEAP[h]`
`⟹ h[CONST n]i r[h] HEAP[i].`

Standard suspensions will now also have to be tagged to distinguish them from constructor suspensions.

[2.6] `SUSP r c ENV[e] HEAP[h]`
`⟹ r[h] h[SUSP c e]i HEAP[i].`

The `JUMP` command must now distinguish between the two types of suspension. In the case of a constructor there is no evaluation to be done and execution should terminate. Providing an update scheme for a `JUMP` to a standard suspension, which must be evaluated as before, but not for a `JUMP` to a constructor suspension ensures that execution will halt when the machine encounters a constant, which can only happen when a (well-formed) λ^+-expression has been fully evaluated.

[2.7a] `JUMP r r[h] h[SUSP c e] ⟹ PC[c] ENV[e].`

(An update scheme 2.7b will be defined later.)

4 Strict Abstraction

4.1 Syntax

The strict annotation symbol in the λ^+-calculus is an exclamation mark. It should not be confused with the '!' output convention, introduced in chapter two.

4.2 Intermediate Code

Two new instructions are needed in the intermediate code.

CALL r $[r_1 \ldots r_n]$
: Call the suspension bound to r. Save the current evaluation so that the callee can return upon terminating. Save the values of registers $[r_1 \ldots r_n]$ on the stack.

STORE x
: Bind x in the current environment to the value returned by the preceding $CALL$.

4.3 $\mathcal{E}[\![\cdot]\!]$

A strictly abstracted expression must be evaluated before its value is bound to its variable. Since a function is about to be called the contents of the compile time stack registers must be pushed onto the run time stack. The removal from the stack of the values pushed by $CALL$ is ensured by

continuing the translation of the expression with an empty compile time stack.

$$\begin{aligned}
\mathcal{E}[\![\lambda!x.e]\!]\ \sigma\ (r +\!\!+ \tau) &= CALL\ r\ \tau;\\
&\quad STORE\ x;\\
&\quad \mathcal{E}[\![e]\!]\ \Sigma\ \texttt{[]}\\
\mathcal{E}[\![\lambda!x.e]\!]\ \sigma\ \texttt{[]} &= r := POP;\\
&\quad CALL\ r\ \texttt{[]};\\
&\quad STORE\ x;\\
&\quad \mathcal{E}[\![e]\!]\ \Sigma\ \texttt{[]}
\end{aligned}$$

where, again, Σ is the list of all available registers.

4.4 $\mathcal{T}[\![\cdot]\!]$

The machine configuration for these new commands is simple.

$$\begin{aligned}
\mathcal{U}[\![CALL\ r\ [r_n \ldots r_1]]\!] &= (\texttt{PUSH r}_n\\
&\qquad \vdots\\
&\qquad \texttt{PUSH r}_1\\
&\qquad \texttt{CALL r}, \qquad \epsilon)
\end{aligned}$$

$$\begin{aligned}
\mathcal{U}[\![STORE\ x]\!] &= (\texttt{POP RO}\\
&\qquad \texttt{BIND}\ \sigma(x)\ \texttt{RO}, \quad \epsilon)
\end{aligned}$$

where σ is, again, the indexing function for identifiers in the environment.

A `STORE` command could have been introduced as the translation of $STORE$, but at this level it is observable that results are always returned via the stack. The use of `RO` to hold the value being fetched is safe, since $STORE$ is always the first command after a $CALL$, which has just cleared all registers by pushing them to the run time stack. Execution must no longer terminate when a constant is encountered, but rather the machine must return to the caller, if any, and place a pointer to the constructor suspension on the stack for the use of the caller. This will be dealt with in the next section.

4.5 Machine Instructions

Strict abstraction will be presented in two steps. Before general strict abstraction is considered, the evaluation method for first order functions will be presented. A function returning a basic value will terminate at a `JUMP` to a constructor expression. Execution must then return to the caller. The caller must therefore have created a suspension containing the continuation address and the current environment. The address of this suspension is passed, like any other argument, via the stack.

```
[4.1a] PC[p] p[CALL r]q r[h] h[SUSP c e]
                                  ENV[f] HEAP[i] SP[t]
   ⟹ PC[c] ENV[e] i[SUSP q f]j
                                  HEAP[j] s[i]t SP[s].
```

This ensures that the requisite information is available for the `JUMP` command.

[2.7b] JUMP r r[h] h[CONST] SP[s] s[i] i[SUSP c e]

$\Longrightarrow$ PC[c] ENV[e] s[h].

(Note that this scheme does not replace scheme 2.7a, but is an alternative to it.)

The key to the specification of the CALL command is the locator expression i[SUSP q f]j on the right hand side of scheme 4.1a. Here CALL has encountered a suspension (locator expression h[SUSP c e] on the left hand side). This suspension must now be evaluated by moving c, the pointer to its code, to the programme counter, and moving its environment pointer e to the current environment register. Execution of the current function must continue upon completion of the call. A new suspension representing the current state of evaluation of the function is therefore created — the locator expression i[SUSP q f]j — and its address is pushed onto the stack.

If CALL encounters a constructor suspension this can be returned immediately.

[4.1b] CALL r r[h] h[CONST] SP[t] $\Longrightarrow$ SP[s] s[h]t.

Clearly this mechanism will only work if functions always return basic values — i.e. functions are always fully evaluated.

The second step in the presentation of strict abstraction is allowing partial evaluation of functions — making higher order functions possible. For example, in the FLIP programme

```
add x y  =  x + y.
 f !g x  =  g x.

f (add 2) 3.
```

evaluation of add 2, forced by f being strict in its first argument, will yield a suspension representing the function $(\lambda x.add\ 2\ x)$. Care must be taken that the function called does not pop more arguments from the stack than it was given. A call must simulate the creation of a new stack, with a new stack bottom. To this end a CALL notes the current top of the stack on a tripwire stack, addressed via the register TP. The value at the top of the tripwire stack is therefore the address of the bottom of the "new" stack.

[4.1a] PC[p] p[CALL r]q r[h] h[SUSP c e]
ENV[f] HEAP[i] SP[t] TP[v]

$\Longrightarrow$ PC[c] ENV[e] i[SUSP q f]j
HEAP[j] s[i]t SP[s] TP[u] u[s]v.

POP must now check for stack underflow against the value stored at the top of the tripwire stack. If the stack has reached the tripwire it is time to halt evaluation of the current suspension and return to the caller. A new suspension must be created containing the partially evaluated expression, and this must be returned as the result.

[2.5a] POP r SP[s] TP[u] u[w] s[h]t

$=[s<w]\Rightarrow$ r[h] SP[t].

[2.5b] `POP PC[p] SP[s] TP[u] u[w]v s[h]t`
`h[SUSP c e] ENV[f] HEAP[i]`
$=$`[s=w]`$\Rightarrow$ `PC[c] ENV[e] TP[v]`
`i[SUSP p f]j HEAP[j] s[i].`

Similarly, a `JUMP` encountering a constructor suspension must pop the tripwire stack.

[2.7b] `JUMP r r[h] h[CONST] TP[u]`
`u[s]v s[i]t i[SUSP c e]`
$\Longrightarrow$ `PC[c] ENV[e] TP[v] s[h].`

The update schemes are now probably reaching the limit of their readability. Chapter seven defines a macro-like mechanism which can abstract away from detail, and greatly increase the readability of update schemes.

5 Predefined Functions

In the discussion of basic values in section 3 values of the type *integer* were used as a typical example. Here the addition function plays the same rôle. Other predefined functions can be treated analogously.

5.1 Syntax

At the surface language level infix expressions such as `x+y` are allowed. However these are desugared to expressions such as $(\lambda!x!y.+\ x\ y)\,x\ y$ at the core language level.

5.2 Intermediate Code

Since the functions are predefined there is no need to generate code for a suspension — this can be taken, for example, from the standard environment.

$r := f$
: Create a suspension for the predefined function f and assign it to register r.

Unsurprisingly, this is similar to the intermediate code generated for basic values. A predefined function can, after all, be considered to be a basic value.

5.3 $\mathcal{E}[\![\cdot]\!]$

The translation of a predefined function is simply that function, though it may help to think of it as the address of a suspension containing the code for that function.

$$\mathcal{C}[\![f]\!] \quad = \quad f$$

5.4 $\mathcal{T}[\![\cdot]\!]$

For any predefined function occurring in a FLIP programme the relevant predefined suspensions must be included in the machine configuration.

In a final implementation all predefined suspensions would probably be included in a standard environment.

$\mathcal{U}[\![r := f]\!] = (\texttt{SUSP r } \pi(f), \epsilon)$
where π is a compile time function returning the address, in the standard environment, of the code for function f.

The code for + could be

```
ADD[POP    R0
    POP    R1
    ADD    R1   R0
    CONST  R0
    JUMP   R0]
```

where $\mathcal{ADD}$ is an internal locator constant. The `JUMP` may look a little strange here, but since the `CONST` will have created a constructor suspension in `R0`, the `JUMP` will simply return to the caller. A single instruction could have been constructed and specified combining the effects of `CONST R0` and `JUMP R0`, but the choice was made to keep the number of instructions low.

5.5 Machine Instructions

The `ADD` command must now access the values in the constructor suspensions addressed by `R0` and `R1`, and perform the addition. As stated above, the `CONST` command will then create a new suspension to contain the result and the `JUMP` will return this to the caller.

```
[5.1] ADD r1 r0 r1[h] h[CONST x] r0[i] i[CONST y]
      ⟹ r0[x+y].
```

6 I/O

6.1 Syntax

The only i/o commands are `read` and `write`. The surface language `write` is compiled to $(\lambda!x.\,\underline{\text{write}}\,x)$ at the core language level. For the sake of simplicity it will be assumed that only integers will be the subject of i/o operations. Note that `write` is strict in its argument. Its only effect is to write the value of its argument to the standard output stream, and is otherwise semantically equivalent to $(\lambda!x.\,x)$.

6.2 Intermediate Code, $\mathcal{E}[\![\cdot]\!]$

The i/o commands `read` and `write` are in fact predefined functions, and are treated as such.

6.4 $\mathcal{T}[\![\cdot]\!]$

Since `read` has arity 0 its predefined code is

```
READ[READ   R0
     CONST  R0
     JUMP   R0]
```

Similarly, `write` has arity 1 and predefined code

```
WRITE[POP    R0
      WRITE  R0
      JUMP   R0]
```

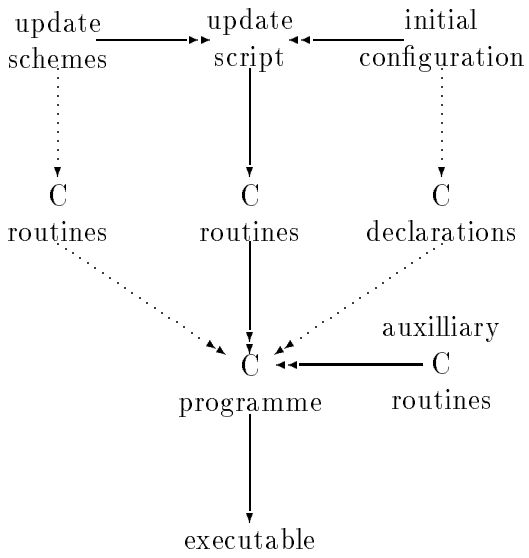


Figure 3: Compilation scheme for FLIP programmes

6.5 Machine Instructions

The constant locators `?` and `!` appearing in these schemes are the input and output buffers respectively. It is assumed that the i/o channels perform the necessary conversions.

[6.1] `READ r ?[n]` $\Longrightarrow$ `r[n].`

[6.2] `WRITE r r[h] h[CONST n]` $\Longrightarrow$ `![n].`

7 The Implementation

The language and machine presented above have been implemented. A FLIP to abstract machine code compiler has been written. The machine configurations delivered by this compiler are compiled to C by a prototype Update Plan compiler. A complete Update Plan compiler would be capable of compiling update scripts, thus completing the compilation process sketched in figures 1 and 2. However, no such compiler is, as yet, available so the specifications of the commands given here were compiled by hand. This compilation was as close an imitation of machine compilation as possible.

The final stage of the FLIP compilation, update script to executable, is shown in figure 3. Single headed arrows indicate translation steps, double headed arrows simple concatenation. The solid lines in figure 3 indicate the compilation path that would be taken if there were a complete Update Plan compiler, the dotted lines the path actually taken to produce a prototype of the machine presented above. The resulting implementation is sufficiently efficient for realistic prototyping.

The efficiency of the code produced was tested by applying the FLIP compiler to a simple nfib programme. The nfib function is given by

$$\mathsf{nfib}\ n \quad = \quad 1, \text{ if } n < 2$$

$$= \quad 1 + (\mathsf{nfib}\ (n-1)) + (\mathsf{nfib}\ (n-2)),\ \text{otherwise}$$

and has the useful characteristic that $\mathsf{nfib}\ n$ is the number of applications of nfib necessary to compute $\mathsf{nfib}\ n$.

The nfib number of a programme is the value of $\mathsf{nfib}\ n$, divided by the number of seconds required to compute it. The higher the nfib number the better. Declarative language implementations have nfib numbers typically ranging from one of the order 1 [41] to $1,000,000$ [11]. This last value is competitive with imperative languages. The gnu C compiler version 1.36 on a SUN 3/60 running OS 4.1 has an nfib number of $250,000$.

The nfib number for this prototype of FLIP was of the same order as that for Miranda[1] (release 2). On a SUN 3/60, running under OS 4.1, Miranda had an nfib number of $1,200$ and the FLIP implementation, compiled using the gnu C compiler version 1.36 with the optimisation flag set, $1,100$. Simple optimisation of the intermediate code (eliminating unnecessary copying, for example) increased this to $1,400$, and optimisation at the update plan level gave an nfib number of $2,200$.

Though this may not appear impressive, it should not be forgotten that the nfib number for FLIP is for an implementation produced by a general prototyping system, while the nfib numbers given above are for dedicated implementations of specific declarative languages. The important point is that the nfib number is competitive with that of the Miranda interpreter, which experience has shown to be efficient enough for realistic programming. An nfib programme written directly in update schemes had an nfib number comparable to that of a programme written in C.

[1] Miranda is a trademark of Research Software Ltd.

Two important aspects of Update Plans are their *implementability* and their *analysis*. Implementational aspects such as backtracking are dealt with in chapter ten, here "implementability" is intended to cover the more fundamental aspect of whether it is possible to implement a given update plan at all, in particular whether or not there is a manageable degree of ambiguity. This is related to the types of objects present in the update plan. Typing is also relevant to the analysis of memory use in update plans. Simple analysis generalises the requirements for certain standard types of memory structure such as stacks, heaps, input and output streams and static stores, thus providing a simple taxonomy of these structures.

Typing is introduced in section 1 and presented in detail in section 2. Section 3 discusses the relation between typing and ambiguity. The rôle of typing in detecting memory structures is covered in section 4, which defines some common memory structures. In section 5 *programme stores* are defined. The semantic properties of these, and other memory use paradigms, are discussed in chapter six.

1 Introduction

The basic type in Update Plans is the locator. In fact all objects appearing in an update plan are considered to be locators. One immediate consequence of this is that every type has a countable carrier set. In fact each type is the carrier of a unique countably infinite set. This does not exclude the use of update plans for specifications involving even simpler types, since any countable type can be realised by means of an injective function from its carrier to some set of locators. A disciplined use of this facility makes it possible to use Update Plans for very low level specifications, for example at bit sequence level by using reserved registers for conversion from bit sequences to locators.

The set of locators can also be partitioned into disjunct sets, or stores, as discussed in chapter three. Membership of one of these stores can also be considered as a form of typing, so the basic type *locator* is

actually a finite set of types. In this view any new type may, if its carrier set is countable, be defined as a new store having the elements of that carrier set as locators, and in this chapter 'type' and 'store' may be used interchangeably, as the context requires.

2 Typing

New types may be defined by combining existing types using any of the standard operators of regular expressions. Such a declaration is said to define a *type alias*. While some typing information may be derived automatically [45] it is the update plan writer's responsibility to ensure that the plan is consistently typed. The writer must indicate which stores are present in the machine being specified. Store names are lower case words with an initial upper case letter, and must, therefore, contain at least two letters, in order to ensure that store names can be distinguished from constants. Stores are declared by listing them between braces:

$$\{\texttt{Stack}, \texttt{Heap}, \texttt{Int}, \texttt{Bool}\}$$

Each store name is said to be a *type primitive*. Type aliases are declared similarly:

$$\{\texttt{Programme} = \texttt{Routine} \cup \texttt{Coroutine}\}$$

Type aliases may not be recursive, either directly or indirectly. This ensures that any type alias can be expressed as a regular expression containing only type primitives. Every object (constant, variable, or expression) appearing in an update plan must be typed. For some objects this may be done implicitly. It is, for example, assumed that a number is of type `Int`, unless indicated otherwise. Each symbolic constant is considered to have its own unique type, again unless indicated otherwise. Other objects — expressions for which no type can be determined automatically — must have their type indicated. This can be done in two ways:

- by means of a global declaration

  $$\texttt{h} :: \texttt{Heap}.$$

  A global type declaration is valid throughout an update plan, unless overridden by a cast;

- or by *casting* a term within an update scheme,

  $$\ldots\ (\texttt{h} :: \texttt{Heap})[\ldots]\ldots$$

  such a cast determining only the type of the term to which it is applied. Any variables in the term share their value, coerced if necessary, with other occurrences of the variable in the update scheme. A cast is only properly defined if all necessary coercion mappings are defined.

A term given by a regular expression has the type defined by the same regular expression with all its constituent terms replaced by their types. The formal syntax of type declarations is obtained by modifying the grammar in chapter two as follows:

- Add the production rules:

  ⟨item⟩ → ⟨store declaration⟩ **.** |
  ⟨type declaration⟩ **.**

- Add also the production rules:

  ⟨term⟩ → **(** ⟨term⟩ **::** ⟨store structure⟩ **)**

  ⟨type declaration⟩ →
  {⟨term⟩ **,**}$^{+}$ **::** ⟨store structure⟩

- and the production rules:

  ⟨store declaration⟩ → **{** {⟨store⟩ **,**}* **}**

  ⟨store⟩ →
  ⟨store name⟩ **=** ⟨store structure⟩ |
  ⟨store name⟩

A *store name* is a lower case word with a leading upper case letter, and a *store structure* is a regular expression over the set of store names.

Store declarations and the store structure parts of casts can also be seen as defining a context free grammar, known as the *type grammar*, having type names as its terminal symbols, and type aliases as its nonterminals. Some additional nonterminals may need to be introduced for implicit type aliases in casts. If a "pure" context free grammar is required yet more nonterminals may be needed to replace closure operators. Defining a context free grammar when type aliases can be defined in terms of regular expressions may seem an overkill, but the type grammar is useful as the basis for the *archetype grammar*, introduced in chapter seven.

Type declarations and casts associate a symbol from the type grammar to each variable appearing in the update plan.

3 Ground Expressions

An absolute requirement for implementability of Update Plans is that *bounded nondeterminism* be avoided. An update scheme exhibits unbounded nondeterminism if it may instantiate to infinitely many update rules, as in example 1 on page 42. Quite apart from the practical difficulties of providing any realistic implementation of Update Plans with unbounded nondeterminism, there are also serious theoretic reasons for avoiding it, since the presence of unbounded nondeterminism complicates any formal semantics [3, 4, 65].

Example 1

Consider the update scheme

$\Longrightarrow \mathtt{a[0]}$.

This update scheme will nondeterministically set the contents of some cell in the store to zero. If included in an update plan it will ensure that the store eventually (at time = ∞) contains nothing but zeros. Clearly this degree of nondeterminism cannot be permitted.

Unbounded nondeterminism is avoided by requiring every expression in an update scheme to be *ground* or *semi-ground*. Intuitively a ground expression is one for which a variable-free expression can be derived, possibly by instantiation of variables with respect to the current configuration; a semi-ground expression one for which a finite number of such expressions can be derived. A *statically semi-ground* term is one for which a variable free expression can be derived without reference to a configuration. A formal definition is given below.

Example 2

Given typing

$\mathtt{x} :: \mathtt{Some\_type}$.
$\mathtt{xs}, \mathtt{ys}, \mathtt{zs} :: (\mathtt{Some\_type})^*$.

in the update scheme

$\mathtt{A[x]b[xs]c[ys]D[zs]e} \Longrightarrow$.

A and D are ground, by virtue of being constants; b is also ground, being equal to $\mathtt{A} + 1$; c is semi-ground ($\mathtt{c} \in \{\mathtt{A} + n \mid 1 \leq n \leq \mathtt{D} - \mathtt{A}\}$); and e is non-ground, since the length of the object represented by zs cannot be determined. Similarly x is ground, xs and ys are semi-ground, and zs is non-ground.

Semi-ground terms are defined by the following rules (rule 5 is explained on page 43, rule 7 in chapter seven):

1. A ground term is semi-ground.
2. If α and β are semi-ground, and $\alpha[\gamma]\beta$ is a locator expression then:
   - if $\alpha[\gamma]\beta$ is on the left hand side of the update scheme, γ is semi-ground;
   - if $\alpha[\gamma]\beta$ occurs only in the right hand side of the update scheme, $|\gamma|$ is semi-ground ($|\cdot|$ is the length operator).

3. If α and $|\gamma|$ are semi-ground then so is β in the expressions $\alpha[\gamma]\beta$ and $\beta[\gamma]\alpha$.

4. The tuple $(\alpha_1, \ldots, \alpha_n)$ is semi-ground if and only if $\alpha_1, \ldots, \alpha_n$ are semi-ground.

5. If f is a well defined mapping for objects of type τ, and α is of type τ, then $f\ \alpha$ is semi-ground.

6. If the language of the nonterminal from the type grammar associated with an object γ is finite then $|\gamma|$ is semi-ground. If the language contains only one element $|\gamma|$ is ground.

7. If $m(\ldots)$ is an archetype call, and the expressions appearing at the parameter positions in some subset p of the archetype's parameter position set are all semi-ground, then so are all expressions appearing at parameter positions in $\mathcal{G}_m(p)$, where $\mathcal{G}_m$ is the grounding function of m. See chapter seven for definitions of 'parameter position set', and $\mathcal{G}_m$.

Note that no use is made of the guard in determining grounding. Any semi-ground term can be instantiated without reference to the guard.

Rules 1 to 3 derive grounding information from the structure of the update scheme under consideration. The asymmetry in rule 2 is due to the fact that a value can be found for a γ on the left hand side by examination of the current configuration, while this cannot be done for a γ which does not appear on the left hand side. All mappings involved in rule 5 are considered to have one argument, and rule 4 is provided for tupling and untupling arguments of mappings. In rule 5 a well defined mapping from objects of type τ to objects of type τ' is one in which every object of type τ' maps to a finite number of objects of type τ'. The set of well defined mappings available may differ from implementation to implementation. An implementation specification must detail which mappings are defined. These must include the basic arithmetic operators, the length operator, and the concatenation operator and its inverse. It is by application of this last that `c` in example 2 is semi-ground, as shown in figure 1. Rule 6 makes use of typing information to determine whether an object has a finite number of possible lengths. Rule 7 is provided for completeness, the terminology used will be explained in chapter seven.

In contrast to Meijer's definition of traceability [53] the grounding rules above preclude the use of C-like strings, in which the end of the sequence is indicated by some special value. Such structures can however easily be defined by way of the archetype mechanism, introduced in chapter seven. Anticipating chapter seven, such a structure could be defined by

$$\begin{array}{lcl} \texttt{string([])} & = & \texttt{0|}. \\ \texttt{string(c s)} & = & \texttt{c string(s)|} \Leftarrow \texttt{[c} \neq \texttt{0]} \Rightarrow . \end{array}$$

4 Memory Structures

The typing system also makes it possible to detect certain common types of addressing structure. This is achieved by means of a store-by-store

semi-ground terms	rule	newly semi-ground terms	mapping
—	(1)	A, D	
—	(6)	|x|	
A, |x|	(3)	b	
A, b	(2)	x	
b, D	(2)	xs ys	
xs ys	(5)	xs, ys	$++^{-1}$
xs, ys	(5)	|xs|, |ys|	$\|\cdot\|$
b, |xs|	(3)	c	
b, c	(2)	xs	
c, D	(2)	ys	

Figure 1: Grounding terms in example 2

analysis of the update schemes in the update plan, and the way in which they are addressed. The aim of the analysis is to detect stacks, and variants and combinations of stacks, which define other common types of memory structure. Knowledge of addressing structures can be useful in determining semantic properties of update plans, as demonstrated in chapter six. The classification discussed below is summarised in figures 1 and 2 in appendix IV.

4.1 Structured Access

In the context of this analysis a locator expression will be called an *access* of the store in which it occurs. An access on the left hand side of an update scheme will be called a *read access*, and on the right hand side a *write access*. An access is *direct* if one of its locators is contained in a register. The register is said to address the access.

If a direct access occurs as part of one of the two patterns

$$R[l]\ l[x]r \Longrightarrow R[l']\ l'[y]r.$$

or

$$R[r]\ l[x]r \Longrightarrow R[r']\ l[y]r'.$$

then the access is said to be structured. In the first case the access is said to *read to the right* and *write to the left*; in the second the access *reads to the left* and *writes to the right*. In both cases, either of the accesses may be empty.

A register R is said to be *dedicated* to a store if its only use in an update plan is to address accesses of that store. A dedicated register may be *read dedicated* if it only addresses read accesses, and *write dedicated* if it only addresses write accesses. The locator in a dedicated register is said to be a *pointer* to the store it addresses.

A store is (read, write) structured if all (read, write) accesses of that store are consistently structured following one of the two patterns above, and each of these accesses has the same dedicated register. A store may be read *and* write structured without being (fully) structured. Only if

the same register is dedicated to read and write accesses is the store fully structured. A store which is read or write structured, but not fully structured, is said to be *semi-structured*.

If a store is read (write) structured, and reads (writes) to the left, those cells to the right of the read (write) pointer are called *lee* cells, cells to the left of the pointer are called *luff* cells. Similarly, if the store reads (writes) to the right cells to the left of the pointer are lee cells, cells to the right luff cells. Lee cells are those cells that have already been read (or written), luff cells have yet to be read (written). In a fully structured store lee and luff cells are determined by the direction in which the store is written.

4.2 Structured Stores

Stacks. A fully structured store is a stack.

Streams. Simpler versions of stacks are those in which there are either no write accesses or no read accesses — input and output streams respectively. These input and output streams are a slight generalisation of those introduced using the '`?`' and '`!`' notation in chapter two in that '`?`' was only used for input streams which read to the right. Similarly, '`!`' was only used for streams which wrote to the right.

4.3 Semi-Structured Stores

Queues. If a store is read and write structured, but not fully structured, it is a *queue*, if it satisfies the following conditions.

- It must read and write in the same direction.
- The read pointer may never point to a luff cell of the write pointer.

The second condition is probably only easy to determine if it is explicitly enforced by a guard at every read access.

Heaps. A simple form of heap is one in which contiguous sections of memory are allocated, and possibly written to, after which any cell in allocated memory may be read. This structure is reflected in the following conditions.

- The store must be write structured.
- Read accesses may only occur in the lee of the write pointer.

The latter condition is guaranteed if, with the exception of the write pointer, any object of the type of the heap, appearing on the right hand side of an update scheme, also appears on the left hand side.

A less restrictive definition of a heap is also possible. This requires that there be a register write dedicated to the heap, and that all other accesses conform to the second condition above.

4.4 Unstructured Stores

Read Only Memory. A read only, or static, store is one in which there are no write accesses.

Write Only Memory. The complement to a read only store is a write only store. In the context of an update plan this could represent, for example, some background storage device, or a graphical display device.

General Purpose Memory. A completely unstructured store represents the most general form of memory use. It is very difficult to derive semantic properties for this type of store.

5 Programme Stores

A special case of read-only memory is a *programme store*. This is the read only memory accessed by `PC` in a command driven plan. Note that the store addressed by `PC` in a command driven store *need not* be read only, and that a programme store therefore need not necessarily be read-only — there is no reason a machine with self-modifying code could not be specified. In the remainder of this thesis, however, all command driven update plans presented will have read-only programme stores, since this seems the most natural style of specifying instruction sets. Extending the relevant results to writeable programme stores should not present unsurmountable difficulties.

If, for a given command driven update plan, the register `PC` always addresses the same store, and this store is read only, then the store involved is said to be a *programme store*. The contents of a programme store are said to form a *programme*. A programme, together with the register `PC` and its contents, is said to form the *programme state*.

The semantic properties of programme stores will be analysed in chapter six.

6 Conclusions

This chapter primarily introduces concepts that will be of use in later chapters. These concepts are based on the typing mechanism. The two major concepts are grounding, which is relevant to the archetype mechanism introduced in chapter seven, and the classification of memory use paradigms, the semantic properties of which are discussed in chapter six, and applied in chapter eight.

An obvious question, when confronted with two update scripts, is to what extent the two scripts are "equivalent", or whether one script is a "translation" of the other. In this chapter these concepts will be formalised, paving the way for their application in chapter eight.

Section 1 defines semantic equivalence, with varying degrees of freedom, between update plans. A concise diagrammatical notation for these definitions is introduced in section 2, and applied in section 4, in which some semantic properties of configurations with respect to some given update plan are defined. Section 5 shows how to use the properties defined in section 4, and the more restrictive equivalence properties from section 1, to prove more general equivalence. This method of proof is applied again in chapter eight.

1 Semantic Equivalence

In this section various semantic relations between update plans will be defined. The properties fall into two classes, *equivalences* and *translations*. These classes can be subdivided pairwise along three axes, as suggested in figure 1, with the default values being at the corner where the axes meet — e.g. 'partial translation' stands for 'general transitive partial translation', and 'restricted local equivalence' for 'restricted full local equivalence'. The axes are characterised as follows (a fuller definition is

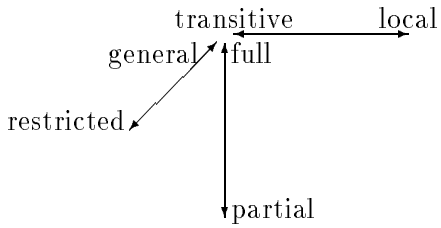


Figure 1: Types of semantic equivalence

given below):

full $\leftrightarrow$ **partial** A semantic property is full if it applies to the whole of any configuration, and partial if it only applies to some well defined subset, for example some set of stores within configurations.

transitive $\leftrightarrow$ **local** Local properties need only hold across one update, while transitive properties are, naturally, transitive.

general $\leftrightarrow$ **restricted** A property is general if no restrictions (other than those implicit in its further classification) are imposed, and restricted if there *are* additional restrictions.

Equivalence and translation are defined in section 1.1. Section 1.2 introduces the full and partial variants of these properties, and section 1.3 the local and transitive forms. General and restricted properties will not be discussed until section 3, after the introduction of box diagrams in section 2.

1.1 Equivalences and Translations

In the context of these properties two update plans are *equivalent* if they have the same effect on configurations, i.e. if any development of one plan can be reached by the other, and vice versa. Update plans p_1 and p_2 are equivalent if, for any configuration c, it is true that $(c \Rightarrow^*_{p_1} c') \Longleftrightarrow (c \Rightarrow^*_{p_2} c')$. The c's in this relation need only be identical up to isomorphism. Any such isomorphism is assumed given, and is implicit in all definitions of (semantic) equivalence. The (semantic) equivalence symbol is '$\equiv$'.

An update plan is a translation of another if the first can reach any configuration that the second can. The reverse need not be true. Plan p_1 is a translation of p_2 if

$$(c \Rightarrow^*_{p_1} c_1) \Longrightarrow \exists c'[(c_1 \Rightarrow^*_{p_1} c') \wedge (c \Rightarrow^*_{p_2} c')]$$

and

$$(c \Rightarrow^*_{p_2} c') \Longrightarrow (c \Rightarrow^*_{p_1} c'),$$

again for any configuration c. The translation symbol is '$\rhd$', thus, above, $p_1 \rhd p_2$. The terminology is inspired by compilers of high level languages in which one high level construct is translated into many low level constructs. The low level translation may pass through many configurations invisible at the higher level.

Where, in the definition of equivalence, two configurations were considered equal modulo isomorphism, in translation two configurations are "equal" modulo a homomorphism from configurations on the right hand side of the translation (in this case p_2) to configurations on the left hand side (p_1). This implies that the definition is only required to hold for configurations of p_1 that are images of configurations of p_2. (A configuration is said to be a configuration of update plan p if it is intended for inclusion with p in an update script.)

1.2 Full and Partial Properties

Often a specification will include "important" and "unimportant" stores. Sometimes a (possibly dynamically determined) part of an individual store may be "important", and the rest "unimportant". It is often useful to restrict consideration of semantic equivalence to relevant sets of (portions of) stores. In partial equivalence (or translation), two developments c_1 and c_2 are considered to be the "same" if, for some set C of "important" cells, they can be written as $c \cup l_1$ and $c \cup l_2$ respectively, again modulo an iso- or homomorphism, with $\mathsf{Dom}\ c \subseteq C$ and $\mathsf{Dom}\ l_i \cap C = \emptyset$, for $i \in \{1, 2\}$. The update plans are said to be equivalent *within* (or to be a translation within) C. In discussions involving both partial properties and semantic irrelevance (defined in section 4) it will be useful to be able to refer to the sets of "unimportant" stores. The terminology *outside* will be used as the converse of 'within' — i.e. if two plans are equivalent within a set C, they can also be said to be equivalent outside the complement of C.

Partiality is indicated by superscripting the equivalence or translation symbol with the set within which the property holds, e.g. $p_1 \equiv^C p_2$.

1.3 Transitive and Local Properties

Any of the above properties is local if it holds for configurations derived in zero or one update, but not necessarily for configurations derived in any number of steps. For example, full local equivalence is defined by $(c \Rightarrow_{p_1}^{(0|1)} c') \iff (c \Rightarrow_{p_2}^{(0|1)} c')$, (where $c \Rightarrow^{(0|1)} c'$ is equivalent to $c \Rightarrow c' \vee c = c'$) and full local translation by

$$(c \Rightarrow_{p_1}^{(0|1)} c_1) \Longrightarrow \exists c'[(c_1 \Rightarrow_{p_1}^{*} c') \wedge (c \Rightarrow_{p_2}^{(0|1)} c')]$$
$$\wedge$$
$$(c \Rightarrow_{p_2} c') \Longrightarrow (c \Rightarrow_{p_1}^{*} c').$$

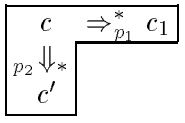
Locality is indicated by subscripting the equivalence or translation symbol with a '1'.

It is obviously far easier to prove local properties than transitive properties. Section 4 defines new properties that can be used to derive transitive properties from their local counterparts.

2 Box Diagrams

The properties above all take the form of existential dependencies. This section presents a diagrammatical notation, known as *box diagrams*, for such dependencies. Existence of any part of the "boxed" part of a box diagram guarantees the existence of the whole diagram. Here 'part' means a set of objects (configurations) from the box diagram, plus any relations specified between the elements of the set. The part of the diagram from which the existence of the remainder of the diagram is deduced is called the *seed*.

For example, the full transitive translation property, $p_1 \rhd p_2$, can be expressed by the following box diagram.



The seeds in this box diagram are $\{c\}$, $\{c_1\}$, $\{c'\}$, $\{c, c_1 \mid c \Rightarrow^*_{p_1} c_1\}$, $\{c, c' \mid c \Rightarrow^*_{p_2} c'\}$, $\{c_1, c'\}$ and $\{c, c_1, c' \mid c \Rightarrow^*_{p_1} c_1 \wedge c \Rightarrow^*_{p_2} c'\}$. The essential existential dependencies are those already given in section 1.3, namely

$$\begin{array}{l} \quad \forall c, c_1[(c \Rightarrow^*_{p_1} c_1) \Longrightarrow \exists c'[(c_1 \Rightarrow^*_{p_1} c') \wedge (c \Rightarrow^*_{p_2} c')]] \\ \wedge \\ \quad \forall c, c'[(c \Rightarrow^*_{p_2} c') \Longrightarrow \exists c_1[c \Rightarrow^*_{p_1} c_1 \Rightarrow^*_{p_1} c')]] \end{array}$$

but it also implies the trivial dependencies (the first, for example, is satisfied by $c_1 = c' = c$)

$$\begin{array}{l} \quad \forall c[\exists c_1, c'[(c \Rightarrow^*_{p_1} c_1 \Rightarrow^*_{p_1} c') \wedge (c \Rightarrow^*_{p_2} c')]] \\ \wedge \\ \quad \forall c_1[\exists c, c'[(c \Rightarrow^*_{p_1} c_1 \Rightarrow^*_{p_1} c') \wedge (c \Rightarrow^*_{p_2} c')]] \\ \wedge \\ \quad \forall c'[\exists c, c_1[(c \Rightarrow^*_{p_1} c_1 \Rightarrow^*_{p_1} c') \wedge (c \Rightarrow^*_{p_2} c')]] \\ \wedge \\ \quad \forall c_1, c'[\exists c[(c \Rightarrow^*_{p_1} c_1 \Rightarrow^*_{p_1} c') \wedge (c \Rightarrow^*_{p_2} c')]] \\ \wedge \\ \quad \forall c, c_1, c'[((c \Rightarrow^*_{p_1} c_1) \wedge (c \Rightarrow^*_{p_2} c')) \Longrightarrow (c_1 \Rightarrow^*_{p_1} c')] \end{array}$$

It is no accident that the essence of the box diagram is that part generated by non-trivial seeds — i.e. seeds which themselves contain some restriction on their constituent configurations. When reading box diagrams it is advisable to first consider such seeds.

Box diagrams can be used for a simple form of proof by diagram chasing. Any box diagrams with overlapping boxed parts may be combined to form new box diagrams. See section 4.1 for a simple example of diagram chasing, and section 5 for a more complex example.

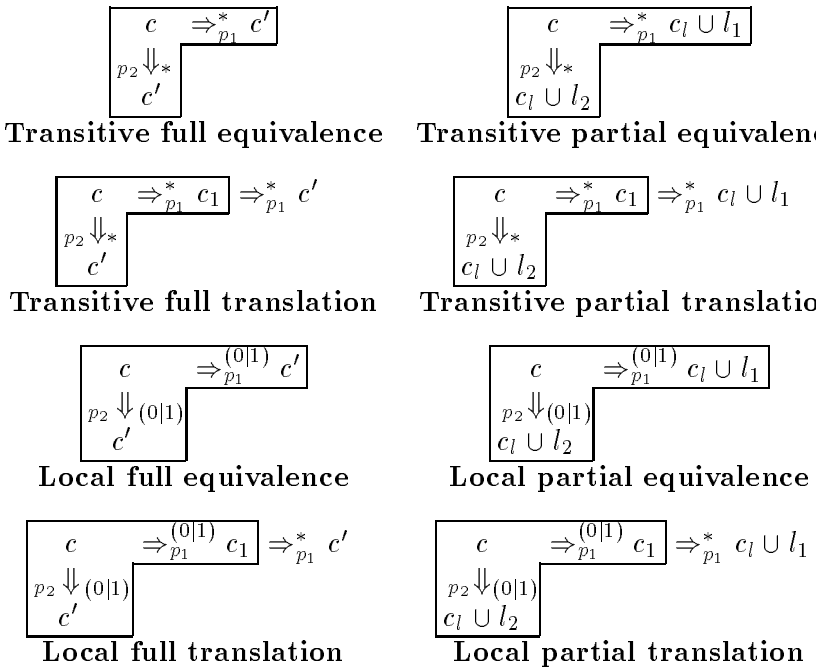
The eight properties given by the combination of the "equivalence $\leftrightarrow$ translation" dichotomy and the "local $\leftrightarrow$ transitive" and "full $\leftrightarrow$ partial" axes in figure 1 are defined by the box diagrams in figure 2.

3 Restricted and General Properties

The third axis in figure 1 covers the possibility of adding extra conditions to a box diagram. Such extra restrictions apply to objects occurring within the box part of a diagram. If any such object is part of a seed the relevant restriction(s) must be included in the preconditions for the existence of the rest of the diagram. A typical restriction might require, for example, a stack pointer not to point to the bottom of a stack.

4 Semantic Irrelevance

The converse of the partiality above is the concept of *semantic irrelevance*. The partiality of partial equivalence (or translation) is a statement that part of the configuration is of no interest in a (final) development, though it may have had an effect on the derivation of that development. A typical example, in the specification of some machine, would be a set of temporary registers — the contents of temporary registers are almost certainly relevant to the semantics at intermediate stages, but of no interest once a final development is reached. Semantic irrelevance identifies



If the partiality of the above properties is outside L then, in all diagrams, $\mathsf{Dom}\ c_l \cap L = \emptyset$ and $\mathsf{Dom}\ l_i \subseteq L$ for $i \in \{1,2\}$.

Figure 2: Semantic properties of update plans

part of the configuration as having no effect on the semantics with respect to the rest of the configuration, although it may well be of interest. An output stream is a good example of a semantically irrelevant store — an output stream does not affect the development of an update plan, but output is almost certainly a feature of interest in the semantics.

While equivalence and translation are relations between update plans, irrelevance is a property of sets of locators relative to a given plan.

4.1 Irrelevance

For some update plan p, the set of locators I is semantically irrelevant to configuration c if the contents of the cells addressed by locators in I are irrelevant to developments of c, i.e.

$$\begin{array}{|l} c_i \cup i_1 \Rightarrow^* c_i' \\ c_i \cup i_2 \Rightarrow^* c_i' \end{array}$$

where $\mathsf{Dom}\ i_j \subseteq I$ for $j \in \{1,2\}$ and $\mathsf{Dom}\ c_i \cap I = \mathsf{Dom}\ c_i' \cap I = \emptyset$. The existence of either of the derivations in the above box diagram guarantees the existence of the other. Again local, partial and restricted versions can be constructed. Local partial irrelevance is, for example, defined by

$$\begin{array}{|l} c_i \cup i_1 \Rightarrow^{(0|1)} c_l \cup l_1 \\ c_i \cup i_2 \Rightarrow^{(0|1)} c_l \cup l_2 \end{array}$$

with $\mathsf{Dom}\ c_i \cap I = \emptyset$, $\mathsf{Dom}\ c_l \cap L = \emptyset$, and $\mathsf{Dom}\ l_j \subseteq L$ for $j \in \{1,2\}$. L will typically be a subset of I.

4.2 Store Structure and Semantic Irrelevance

It is easy to prove the following facts. (See chapter five for a definition of lee and luff cells.)

- a write-only store is semantically irrelevant
- lee cells in a read-only store are semantically irrelevant
- luff cells (with respect to the write pointer) are semantically irrelevant in queues, stacks and heaps

The last case is a special case of the general rule that, if it is possible to prove for a write structured store that a read access will always take place in the lee of the write pointer, then the luff cells of that write pointer are semantically irrelevant.

Note that the first property is general (a write-only store is a write-only store, irrespective of the current configuration), while the remaining properties are restricted — precisely which cells are lee or luff cells depends on the current configuration.

4.3 Diagram Chasing

In section 5.1 transitive translation properties will be derived from local translation and local and transitive irrelevance by diagram chasing. As preparation for this proof, a simple example of proof by (box) diagram chasing will be presented. It will be shown how to establish transitive irrelevance properties from their local counterparts.

Take, for example, transitive partial irrelevance. This can be derived from local partial irrelevance by inductive diagram chasing. Let I be the irrelevant set, and L be the set outside of which the irrelevance holds. Assume furthermore that L is a subset of I. The local irrelevance property is then expressed by

$$\boxed{\begin{array}{l} c_i \cup i_1 \Rightarrow^{(0|1)} c_l \cup l_1 \\ c_i \cup i_2 \Rightarrow^{(0|1)} c_l \cup l_2 \end{array}}$$

with $\mathsf{Dom}\ c_l \cap L = \emptyset$ and $\mathsf{Dom}\ l_j \subseteq L$ for $j \in \{1,2\}$. Since L is a subset of I, $c_l \cup l_1$ can be rewritten as $c_i' \cup i_1'$ with $c_i' \cap I = \emptyset$ and $i_1' \subseteq I$, by "transferring" part of c_l to i_1'. Similarly $c_l \cup l_2$ can be rewritten as $c_i' \cup i_2'$, and the local irrelevance property can be applied again, giving

$$\boxed{\begin{array}{lll} c_i \cup i_1 & \Rightarrow^{(0|1)} & \boxed{\begin{array}{l} c_i' \cup i_1' \\ c_i' \cup i_2' \end{array}} \Rightarrow^{(0|1)} \quad c_l'' \cup l_1' \\ c_i \cup i_2 & \Rightarrow^{(0|1)} & \qquad\qquad\ \Rightarrow^{(0|1)} \quad c_l'' \cup l_2' \end{array}}$$

The diagram can be rewritten as

$$\boxed{\begin{array}{l} c \Rightarrow^{(0|1|2)} c_l' \cup l_1' \\ c \Rightarrow^{(0|1|2)} c_l' \cup l_2' \end{array}}$$

and states that for any l_2', if $c_l' \cup l_1'$ can be derived from c in 0,1 or 2 steps $c_l' \cup l_2'$ can also be derived from c in 0,1 or 2 steps, and vice versa. Proofs

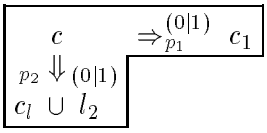
for other flavours of irrelevance are similar. Transitive irrelevance does not necessarily imply local irrelevance.

5 Deriving Transitive from Local Properties

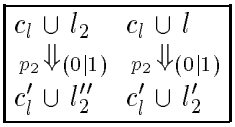
Diagram chasing can be used not only to derive transitive irrelevance properties, but also, given an appropriate irrelevance property, to derive transitive translation (or equivalence) from the corresponding local property.

5.1 Methodology

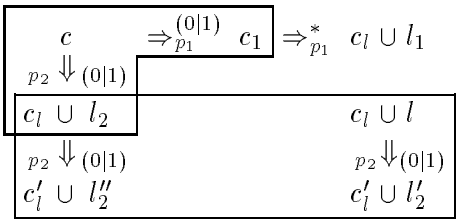
The methodology will again be illustrated by an example. Local partial (outside some set of locators L) translation is defined by:



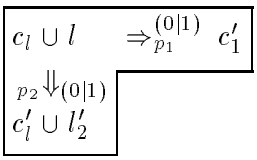
Possibly $c_l \cup l_2$ can be developed further, $c_l \cup l_2 \Rightarrow^{(0|1)}_{p_2} c'_l \cup l''_2$, say. Local irrelevance gives:



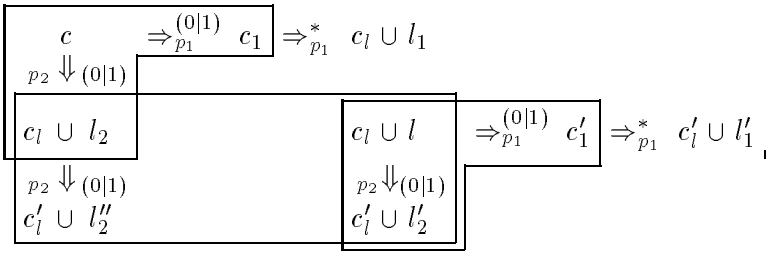
(The diagram has been rotated 90° with respect to the corresponding diagram in section 4.) These two diagrams can be combined to give:



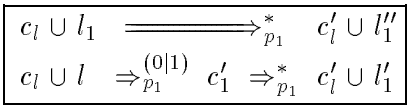
Local partial translation then establishes:



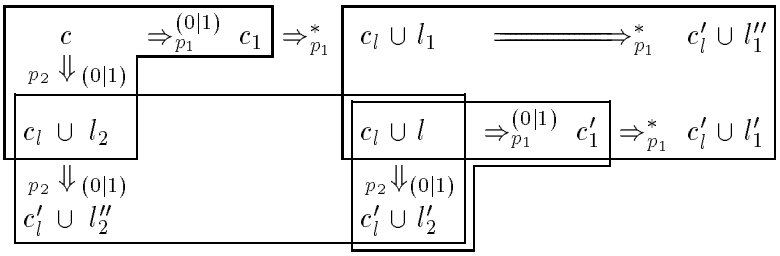
giving:



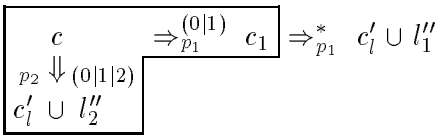
Finally transitive partial irrelevance can be written as:



and combined with the other diagrams to give:



A similar reasoning can be followed given a development $c_l \cup l_1 \Rightarrow^{*}_{p_1} c'_l \cup l''_1$ and it can therefore be concluded that:



thus establishing transitive partial translation from local partial translation and local and transitive partial irrelevance. Since local partial irrelevance implies transitive partial irrelevance, local partial translation and local partial irrelevance suffice to establish transitive partial translation.

The proofs for other flavours are again similar though, again, some boundary conditions may need to be checked.

5.2 Application

The semantic properties, and the proof method, described in this chapter can be applied to proving correctness of programme transformations.

For example, given two command driven update plans, and a "translation" from configurations for the one plan to configurations for the other (usually specifying transformations of the programme store and, implicitly, the contents of the `PC`, and possibly also of some ancillary stores and associated registers, such as a stack and its stack pointer) it is often the question whether the "translation" is a translation in the sense defined above.

It is usually relatively simple to prove local (partial) translation, possibly analysing possible configurations and proving restricted translation for separate classes of configuration. It is also usually not too difficult to show (or disprove) irrelevance of those parts of the configuration in which the translation is partial. The construction in section 5.1 can then be used to establish transitive translation. A similar method is applied in chapter eight.

6 Conclusions

This chapter has defined some semantic properties of update plans and scripts, and shown how to derive more general properties from more restricted ones. An initial description of an area of application of the subject matter of this chapter has been given. The methods of this chapter will be applied again in chapter eight.

Update Plans constitute a high level language for the specification of low level activities. They combine concrete and detailed low-level descriptions of data structures with an abstract and high-level algorithmic specification formalism. This chapter adds a potent macro-like mechanism to Update Plans, known as *archetypes*, which increases the expressive power of the formalism by facilitating abstraction away from unnecessary detail, and making it possible to specify concrete machines in an easy and natural way. The archetype mechanism is also eminently suited for the specification of parallel architectures.

Archetypes resemble parameterised macros and represent substructures of update schemes, making a form of structured programming possible. The parameter mechanism is similar to that of attribute grammars[18, 23, 39, 40, 44, 49], with parameters being either inherited or derived (synthesised).

1 Archetypes

Update Plans have already been used for the specification of abstract machines. The complexity of such machines is usually to be found in their command set, rather than in their addressing modes. Each command tends to have only a small number of addressing modes associated with it — in most cases the instruction determines the addressing mode uniquely. Instructions on concrete machines, on the other hand, are often formed by combining elements of two more or less orthogonal sets: the commands and the addressing modes. Specifying each possible combination of command and addressing mode leads to an explosive growth in the number of update schemes needed.

The archetype mechanism greatly increases the expressive power of Update Plans. Using the archetype mechanism complicated pointer structures, families of such structures, and even infinite classes of arbitrarily large structures may be replaced by a single archetype call, thus making it possible to express many update schemes as one.

Archetypes are inspired by macro mechanisms. Their parameter resolution system is purely "macro" in flavour, though their expansion, in particular that of recursive archetypes, may be context driven, i.e. dependent on the configuration in which the macro is expanded. Archetypes bear a degree of similarity to *syntax macros* [46].

Archetypes may be used both in update schemes and in specifications of configurations. The expansion mechanism is slightly more complicated for archetype applications in update schemes, due to the more complex structure of update schemes. In order to give a full presentation the expansion mechanism for archetype applications in update schemes will therefore be given. An archetype application in a configuration is expanded by ignoring any guards and right hand sides in its definition(s), and expanding it as if it were a *left-handed* archetype (see page 59).

1.1 Targeted Use

The primary application of the archetype mechanism is abstracting away the details of addressing modes.

Example 1

Given a suitable set of archetypes defining addressing modes, and the pointer structures associated with these modes, and specifying which of these may be used as source and destination operands (indicated here by `src` and `dst` respectively) the update scheme

$$\texttt{ADD src(x) dst(ea, y)} \Longrightarrow \texttt{ea[x + y]}.$$

will expand to a number of schemes, one for each permissible combination of addressing modes. The source may, for example, be addressed in "register deferred" mode and the destination in "predecrement" mode in which case the relevant expansion of the above scheme could be

```
ADD REGDEF i PREDEC j.
    i[a] a[x]  j[q] p[y]q
⟹
                j[p] p[x + y].
```

This example anticipates some of the syntactic sugar to be introduced in section 2.

A secondary application is the specification of operations on recursive data structures, as illustrated in example 6.

1.2 Syntax

Archetype Definitions. The syntax of archetype definitions is given by the following context free grammar. A guard may not contain an archetype call (this has to do with the structured nature of archetype calls), and a locator may not contain a call of a recursive macro (this has to do with the expansion mechanism).

⟨item⟩ → ⟨archetype definition⟩

⟨archetype definition⟩ →
    ⟨command archetype definition⟩ |
    ⟨basic archetype definition⟩

⟨basic archetype definition⟩ →
    ⟨basic declaration⟩ ⟨basic definition⟩$^+$

⟨basic declaration⟩ → ⟨basic archetype name⟩ ⟨parameters⟩

⟨basic definition⟩ → = ⟨basic body⟩ .

⟨basic body⟩ →
    ⟨configuration⟩ ⟨guard⟩ ⟨configuration⟩ |
    ⟨repeat⟩ ⟨guard⟩ ⟨configuration⟩ |
    ⟨configuration⟩ |
    ⟨text⟩ ‖ ⟨context⟩ ⟨guard⟩ ⟨context⟩ |
    ⟨text⟩ ‖ ⟨repeat⟩ ⟨guard⟩ ⟨context⟩ |
    ⟨text⟩ ‖ ⟨context⟩

⟨command archetype definition⟩ →
    ⟨command declaration⟩ ⟨command definition⟩$^+$

⟨command declaration⟩ → ⟨command archetype name⟩
⟨parameters⟩ ⟨text⟩

⟨command body⟩ → = ⟨command body⟩ .

⟨command body⟩ →
    ⟨context⟩ ⟨guard⟩ ⟨configuration⟩ |
    ⟨repeat⟩ ⟨guard⟩ ⟨configuration⟩ |
    ⟨context⟩

⟨parameters⟩ → ({⟨term⟩ ,}*)

This grammar specifies the sugared syntax of archetype definitions. Syntactic sugar for archetypes is defined in section 2. A *command archetype name* is an upper case identifier, a *basic archetype name* a lower case identifier.

Archetype Calls. The syntax of an archetype call is basically that of a archetype declaration. There is one difference — archetype calls occur in pairs, and the calls may be "coupled" by an index. A term is now an expression built from constants, variables, archetype calls and operators. The following production rules should be added to the Update Plan grammar.

⟨term⟩ → ⟨archetype call⟩

⟨archetype call⟩ → ⟨archetype name⟩ ⟨index⟩-*opt* ⟨parameters⟩

⟨archetype name⟩ →
⟨command archetype name⟩ |
⟨basic archetype name⟩

⟨index⟩ → ⟨number⟩ | **{** ⟨number⟩ **}**

An archetype's definition specifies text to be added to both the left and right hand sides of the update scheme in which it is called. As a consequence any archetype call on the left hand side of an update scheme must have a matching call on the right hand side, and vice versa. Since there may be more than one such pair of calls in some update scheme it is necessary to indicate which calls are coupled. This is done by indexing archetype call with a number, placed between the archetype's name and its parameter list. To distinguish between $\texttt{call}_1(\ldots)$, an indexed call of the archetype `call`$(\ldots)$, and `call1`$(\ldots)$, an unindexed call of `call1`$(\ldots)$, indices may be enclosed in braces (`call{1}`$(\ldots)$). Each archetype–index pair must occur exactly once on both the left and right hand sides of the update scheme in which the archetype is called. The index may be omitted in some cases, as may one member of a pair of a calls, as detailed in section 2. In the following indices will be omitted, where this does not lead to confusion. Example 2 illustrates archetype expansion.

Expansion takes place, conceptually, as follows. For each pair of archetype calls, the archetype calls are replaced by the corresponding expansions, and the left and right hand side contexts are added to the corresponding sides of the update scheme in which the calls occurred. If more than one expansion is possible copies of the update scheme are generated for each possible expansion. See sections 3.2, 4.1 and 4.2 for more details.

Example 2

Consider the archetype definition

`pop(v) = s  s[v]t⟹t.`

Textual expansion (see section 3.1) of this archetype in the update scheme

$\texttt{ADD SP[pop}_1\texttt{(x)] ACC[y]}\Longrightarrow\texttt{SP[pop}_1\texttt{(x)] ACC[x + y].}$

will give

`ADD SP[s] ACC[y] s[v]t⟹SP[t] ACC[x + y].`

The call of `pop(x)` on the left hand side has been replaced by `s`, and that on the right hand side by `t`. The locator expression

`s[v]t` has been added to the left hand side. Parameter resolution (see section 3.2) equates `x` to `v` giving

`ADD SP[s] ACC[y] s[v]t`$\Longrightarrow$`SP[t] ACC[v + y].`

Again, this example anticipates some of the syntactic sugar in section 2.

---

2 Syntactic Sugar

The contexts of archetype definitions inherit syntactic sugar from standard update schemes, enabling unnecessary locators and guards to be omitted. In addition the following archetype specific syntactic sugar is defined. The terminology used is that of the grammar in section 1.2.

2.1 Redundancy in Calls

Left- and Right Handed Archetypes. If, for a given archetype, in all of its definitions, the expansion of the right hand side is empty then the right hand side call of that archetype may be omitted. If all right hand side calls are omitted such an archetype is called a *left handed* archetype. The same holds for empty left hand sides, in which case the archetype is *right handed*. In theory an archetype could be both left and right handed, in which case both calls would be omitted. Since the conditions for omission only guarantee that the expansions of the archetype are empty this could result in "invisible" archetype calls introducing supplementary locator expressions, and is therefore forbidden.

Context Independent Archetypes. Left and right handed archetypes have "singleton" — i.e. unpaired — calls. They are restricted in that a left handed archetype may only be called on the left hand side, and a right handed archetype on the right hand side. An *ambidextrous archetype* is a "singleton" archetype which does not have this restriction. In the definition of an ambidextrous archetype the expansion is separated from the contexts by a vertical line (see example 3). A definition of an ambidextrous archetype `m` is equivalent to a pair of archetype declarations, of $\mathtt{m_l}$ and $\mathtt{m_r}$ say, where $\mathtt{m_l}$ is a left handed archetype and $\mathtt{m_r}$ is right handed.

Example 3

The archetype definition

$\mathtt{m}(\ldots) = text \mid lhs \mathrel{=\![}\ guard\ \mathrel{]\!\Rightarrow} rhs.$

is equivalent to the two archetype definitions

$\mathtt{m_l}(\ldots) = text\ lhs \mathrel{=\![}\ guard\ \mathrel{]\!\Rightarrow} rhs.$

and

$\mathtt{m_r}(\ldots) = lhs \mathrel{=\![}\ guard\ \mathrel{]\!\Rightarrow} text\ rhs.$

where calls of `m` on the left hand side are equivalent to calls of $\mathtt{m}_\mathtt{l}$, and right hand side calls of `m` are equivalent to calls of $\mathtt{m}_\mathtt{r}$.

Since it cannot be determined from an archetype definition where an archetype call appearing in one of its parameters may be applied, only ambidextrous archetypes may appear in parameter expressions.

Omitting Indices. The sugar defined above produces archetypes with "singleton" calls — in contrast to standard archetypes, calls of which are always paired. Since there is no coupling, indexing is redundant and may be omitted. Similarly, if there is only one paired call of some archetype in an update scheme that archetype need not be indexed.

2.2 Redundancy in Definitions

Command Archetypes. If `m` is an ambidextrous archetype and the expansions of all definitions of `m` begin with the same constant then that constant may be used as the archetype name, if it has not already been so used. The remainder of the expansion is then placed between the archetype declaration and the archetype definition symbol. Such an archetype is called a *command archetype*. The command archetype

$$\mathtt{CONST}(\ldots)\ text = lhs =[\ guard\ ]\Rightarrow rhs.$$

is the sugared version of

$$\mathtt{m}(\ldots) = \mathtt{CONST}\ text \mid lhs =[\ guard\ ]\Rightarrow rhs.$$

with calls of `m` replaced throughout the update plan by calls of `CONST`.

Empty Guards and Right Hand Sides. If the guard and right hand side of an archetype definition are both empty then the transition symbol ($\Longrightarrow$) may be omitted.

Shared Formals. Identical archetype declarations may be shared. The period is omitted from the end of each shared definition but the last. The '=' is not omitted.

Don't Care Values. Irrelevant parameters may be replaced by '_', which is the "don't care" symbol.

3 Expansion

The archetype expansion mechanism, inspired by traditional macro expansion, consists of two stages, the *textual expansion stage* and the *parameter resolution stage*, as described in sections 3.1 and 3.2 respectively. In the following the update scheme in which an archetype call occurs will be referred to as *the* scheme of that call. Archetype calls in an expansion or context of an archetype, or as parameter of an archetype, inherit their scheme from the caller.

3.1 Textual Expansion

Both the left and right hand sides of an archetype body consist of two parts, the *expansion* and the *context*. The expansion is the text that actually replaces the archetype call, the left hand side expansion replacing the call on the left hand side of the scheme, and the right hand side

expansion that on the right hand side. The contexts must be added to the call's scheme; again the left hand (right hand) side context is simply textually added to the left hand (right hand) side of the scheme. The guard of the call is joined to the guard of the scheme using conjunction.

More formally, if $(\lambda x.(lhs \ =[\ guard\ ]\Rightarrow\ rhs))(m(p_1,\ldots,p_n))$ is an update scheme containing a call of the archetype $m(p_1,\ldots,p_n)$, and an instantiation of this archetype is

$$m(p_1,\ldots,p_n) = txt_l\ \ sch_l \ =[\ grd\ ]\Rightarrow\ txt_r\ \ sch_r.$$

then the result of textually expanding the archetype in the scheme is

$$lhs' \ =[\ guard \wedge grd\ ]\Rightarrow\ rhs'.$$
$$\text{where}\quad lhs' = (\lambda x.lhs)txt_l\ \ sch_l$$
$$rhs' = (\lambda x.rhs)txt_r\ \ sch_r$$

The definition of textual expansion may seem unduly complicated, but the most obvious alternative, integral *in situ* expansion, while attractive in its simplicity, can lead to unexpected results, as shown in example 4.

Example 4

Consider the following archetype definition, using an ad hoc notation, designed for *in situ* expansion (note that the syntax of this "archetype" does not conform completely to the syntax for archetypes with separate expansion of the text and contexts)

`ref(a) = a[b] b`$\Longrightarrow$.

When called as the left locator of a locator expression, using integral *in situ* expansion, the effect will be as suggested by the archetype's name. The archetype call

... `ref(a)[v]` ...$\Longrightarrow$ ... .

will expand to

... `a[b] b[v]` ...$\Longrightarrow$ ... .

When called as the right locator, however, expansion gives unexpected results. For example

... `[v]ref(a)` ...$\Longrightarrow$ ... .

gives

... `[v]a[b] b` ...$\Longrightarrow$ ... .

Even more problematic is the case of archetype calls within locator expressions.

3.2 Parameter Resolution

If the parameters of archetype calls and definitions were always simply variables parameters would not need to be resolved — the variables could simply be replaced throughout the scheme by the corresponding parameters from the archetype's definition. That a more complicated approach is needed if parameters may be compound expressions can be seen in the following example.

Example 5 (a)

Consider the archetype definition

$$\texttt{bdisp(b + d)} \;=\; \texttt{BDISP r d r[b]}\Longrightarrow.$$

and its application in

$$\texttt{CEQ bdisp}_1\texttt{(x) bdisp}_2\texttt{(x)}\Longrightarrow\texttt{CC[TRUE]}.$$

(Intuitively CEQ is comparing the effective addresses of the two BDISP operands. A more realistic example illustrating the problem could be given, but it would probably lack the simplicity of this slightly forced example.) Let

$$\texttt{bdisp(b}_1 + \texttt{d}_1\texttt{)} = \texttt{BDISP r}_1\ \texttt{d}_1\ \texttt{r}_1\texttt{[b}_1\texttt{]}\Longrightarrow.$$

be the instantiation of the definition which is to be substituted for $\texttt{bdisp}_1\texttt{(x)}$. Consistent substitution of $\texttt{b}_1 + \texttt{d}_1$ for x will give

$$\texttt{CEQ BDISP r}_1\ \texttt{d}_1\ \texttt{bdisp}_2\texttt{(b}_1 + \texttt{d}_1\texttt{)}\ \texttt{r}_1\texttt{[b}_1\texttt{]}\Longrightarrow\texttt{CC[TRUE]}.$$

If $\texttt{bdisp}_2\texttt{(b}_1 + \texttt{d}_1\texttt{)}$ is to be expanded according to the instantiation

$$\texttt{bdisp(b}_2 + \texttt{d}_2\texttt{)} = \texttt{BDISP r}_2\ \texttt{d}_2\ \texttt{r}_2\texttt{[b}_2\texttt{]}\Longrightarrow.$$

an expression $(\texttt{b}_2 + \texttt{d}_2)$ is available which can be substituted for $\texttt{b}_1 + \texttt{d}_1$, but no expressions are available for $\texttt{b}_1$ and $\texttt{d}_1$ individually.

A parameter resolution method is needed for situations such as these. Rather than using a consistent substitution mechanism, parameters are resolved by maintaining a system of equations as rewriting takes place. It should be emphasised that parameter resolution remains a purely textual manipulation. In example 5(a) resolution will yield expressions for $\texttt{b}_1$ and $\texttt{b}_2$, possibly rewriting other expressions in the update scheme, or adding equalities between expressions to the guard.

Example 5 (b)

In this example, the first rewrite will give

```
CEQ BDISP r1 d1 bdisp2(x) r1[b1]
  ⟹CC[TRUE].
```

and the equation $x = b_1 + d_1$. The second archetype can now be rewritten giving

```
CEQ BDISP r1 d1 BDISP r2 d2 r1[b1] r2[b2]
  ⟹CC[TRUE].
```

with equations $\{x = b_1 + d_1, x = b_2 + d_2\}$.

The basis for parameter resolution is grounding, as defined in chapter five. The textual expansion mechanism described in section 3.1 may leave some terms non-ground (e.g. `x` in example 2). The aim of parameter resolution is to derive semi-ground expressions for non-ground terms. The base for resolution is the set of equations generated during textual expansion. This set of equations is called the *resolution set*.

Parameters are resolved iteratively. Conceptually, examination of the scheme of the call, after expansion, determines which expressions are semi-ground. Semi-ground expressions are then found for as yet unresolved variables by applying the equations in the resolution set. The expressions found are then substituted for occurrences of the corresponding variables both in the scheme and in the resolution set, and the process is repeated until all terms are semi-ground. If at any point in this process a non-trivial equation is derived relating two semi-ground expressions this is added to the scheme's guard. This process is slightly complicated by the existence of recursive archetypes, expansion of which is driven by the current configuration, requiring expansion and resolution to be interleaved. Resolution, however, is independent of the expansion mechanism. The parameter resolution mechanism presented here is certainly no more expensive than unification.

Example 5 (c)

Continuing the example

```
CEQ bdisp1(x) bdisp2(x)⟹CC[TRUE].
```

may expand to

```
CEQ BDISP r1 d1 BDISP r2 d2
          r1[b1]       r2[b2]
  ⟹CC[TRUE].
```

with the associated resolution set (semi-ground terms are in bold font) $\{x = \boldsymbol{b}_1 + \boldsymbol{d}_1, x = \boldsymbol{b}_2 + \boldsymbol{d}_2\}$. Substituting $\boldsymbol{b}_2 + \boldsymbol{d}_2$

for x throughout the scheme and the resolution set gives $\{\boldsymbol{b}_2 + \boldsymbol{d}_2 = \boldsymbol{b}_1 + \boldsymbol{d}_1, \boldsymbol{b}_2 + \boldsymbol{d}_2 = \boldsymbol{b}_2 + \boldsymbol{d}_2\}$. The non-trivial equation is added to the scheme's guard, giving

```
CEQ BDISP r1 d1 BDISP r2 d2
          r1[b1]       r2[b2]
  =[ b2 + d2 = b1 + d1 ]=> CC[TRUE].
```

---

4 Recursion

Update schemes are instantiated to update rules by matching them against the current configuration, possibly using type information to determine the number of cells occupied by the value of some variable. This allows variables to represent simple structures; atomic types, arrays, records, etc. It would be useful if there were a mechanism for representing more complicated structures, such as trees. Recursive archetypes provide such a mechanism, at the cost of increasing the complexity of the instantiation mechanism. Archetype definitions can be seen as (embellished) rules in a context free grammar (as defined in section 4.1). Recursive archetypes are therefore expanded during instantiation in a manner akin to grammar driven recognition of strings in context free languages.

Example 6

The recursive archetype definitions

```
tree() = LEAF x==>LEAF x.
tree() = NODE tree1() tree2()
          ==>NODE tree1() tree2().
```

define a binary tree structure. A possible application is in the definition of balancing operations as used for AVL trees. An update scheme for one of these, the "up right" operation, is

```
UPR NODE tree1() (NODE tree21() tree22())
 ==>NODE (NODE tree1() tree21()) tree22().
```

The parentheses around the archetype expansions have been added to improve legibility and are not an essential part of the update scheme.

Recursive archetypes represent infinite hierarchies of expansions and can therefore not be expanded using information based solely on the update plan in which they are defined. Expansion of recursive archetypes is driven by the current configuration. Some guarantee is needed, based on information derived solely from the update plan, that parameter resolution is finite (i.e. that the resolution set is finite) and complete (i.e. that a solution to the resolution set can be found, or shown not to exist).

4.1 Finite Resolution

Parameter resolution will be finite if textual expansion occurs in a finite number of steps. A sufficient condition for this can be given in terms of the *archetype grammar*, a context free grammar based on archetype definitions.

Since it is the left hand side of an update scheme that is matched with the current configuration only information from the left hand sides of archetype definitions may be applied to guarantee termination. Only the expansion of the left hand side is considered, in order to simplify the termination conditions, though extension to cover the whole left hand side may be feasible. Archetypes appearing in an archetype body elsewhere than the expansion of the left hand side may only be recursive if coupled to a call that *is* in the expansion of the left hand side. Archetypes appearing in parameters may not be recursive.

The basis for the archetype grammar is the set of typing rules. The typing rules assign to each variable in an update plan an element, either terminal or nonterminal, of the type grammar, introduced in chapter five. The type grammar defines all types appearing in an update plan. Of particular importance is the distinction between atomic types and sequence types. Objects of an atomic type are assumed to occupy a fixed number of cells, while those of a sequence type occupy a variable number of cells, possibly zero.

In the type grammar atomic types are terminals, and for each non-atomic type there is a nonterminal, the *type generator* of that type, which is the left hand side of a production rule generating exactly the set of possible terminal type expressions for that type. If this is a singleton set its element may be used as type generator.

The archetype grammar is formed by adding production rules, based on the archetype definitions, to the type grammar. Archetype names are added to the set of nonterminals, and any constants appearing in an archetype body to the set of terminals. New production rules are added to the grammar, derived from the archetype definitions by removing the parameter lists from the archetype declarations and deleting all but the left hand side expansion of the archetype bodies. Any expression not soluble at compile time, i.e. any expression containing one or more variables or archetype calls, is replaced by its type generator.

Example 7

Given the type alphabet {**Loc**,**Num**}, the type grammar

$$\langle \text{LocNum} \rangle \rightarrow \mathbf{Loc} \mid \mathbf{Num}$$

and the typing rules $\mathtt{v} \rightarrow \langle \text{LocNum} \rangle$, and $(\mathtt{adr} - \mathtt{reg}_1(\mathtt{r})) \rightarrow$ **Loc**, the archetype grammar corresponding to the archetype

definitions:

$\texttt{op(reg(r))} = \texttt{REG val}_1\texttt{(r)} \Longrightarrow \texttt{val}_1\texttt{(r)}.$
$\texttt{op(v)} = \texttt{BDISP val}_1\texttt{(r) (adr} - \texttt{reg}_1\texttt{(r)) adr[v]}$
$\Longrightarrow \texttt{val}_1\texttt{(r) reg}_1\texttt{(r)}.$

$\texttt{reg(r)} = \texttt{v} \mid \texttt{(R} + \texttt{r)[v]} \Longrightarrow.$

$\texttt{val(v)} = \texttt{v} \Longrightarrow.$
$\texttt{val(v)} = \texttt{op}_1\texttt{(v)} \Longrightarrow \texttt{op}_1\texttt{(v)}.$

is:

$\langle \text{op} \rangle \rightarrow \textbf{REG} \ \langle \text{val} \rangle \mid$
$\quad \textbf{BDISP} \ \langle \text{val} \rangle \ \textbf{Loc}$

$\langle \text{reg} \rangle \rightarrow \langle \text{LocNum} \rangle$

$\langle \text{val} \rangle \rightarrow \langle \text{LocNum} \rangle \mid$
$\quad \langle \text{op} \rangle$

$\langle \text{LocNum} \rangle \rightarrow \textbf{Loc} \mid \textbf{Num}$

Finite resolution is guaranteed if no derivations of the type $\alpha \Rightarrow^+ \alpha$ are possible in the archetype grammar. This ensures that as expansion proceeds the length of the replacement text increases. Since only a finite sequence of cells will be defined at the location of the archetype call only a finite number of expansions is possible. This is the reason recursive archetypes calls may only occur in a term specifying the contents of (a sequence of) cells. In a realistic implementation additional restraints may be imposed on the archetype grammar in order to ensure efficient expansion.

4.2 Complete Parameter Resolution

In chapter five grounding by way of archetype calls was defined in terms of the as yet undefined parameter position set and archetype grounding function. These notions will now be defined.

For any given archetype $m(p_1, \ldots, p_n)$ the parameter position set P_m is defined to be $\{1, \ldots, n\}$. The parameter position set indexes the archetype's parameters. Any archetype call specifying the contents of (a sequence of) cells — i.e. a call not occurring in a locator — also has the implicit parameters ${}_{\lfloor}l$, $l_{\rfloor}$, ${}_{\lfloor}r$ and $r_{\rfloor}$. These implicit parameters are the left and right locators of the left and right hand side call of the archetype, and therefore of the corresponding expansion in the archetype definition. The left locator on the left hand side is indicated by ${}_{\lfloor}l$, the right locator on the left hand side by $l_{\rfloor}$; ${}_{\lfloor}r$ and $r_{\rfloor}$ are interpreted analogously.

For any archetype definition m the grounding function $g : 2^{P_m} \mapsto 2^{P_m}$ can be defined such that $g(p_{in}) = p_{out}$ is equivalent to "if all parameters at positions in p_{in} are semi-ground, then so are all parameters at positions in p_{out}". The value of p_{out} can be determined by applying the grounding rules in chapter five.

Example 8

If archetype neg has definition

$$\texttt{neg(v)} = \texttt{v} \Longrightarrow -\texttt{v}.$$

then its parameter position set is $\{{}_{[}l, l_{]}, {}_{[}r, r_{]}, 1\}$. For this definition some values of g are, assuming v has atomic type

$$\begin{aligned} g(\{r_{]}\}) &= \{{}_{[}r, r_{]}\} \\ g(\{l_{]}\}) &= \{{}_{[}l, l_{]}, 1\} \\ g(\{{}_{[}l, {}_{[}r\}) &= P_{neg} \end{aligned}$$

The function g is extended to cover all definitions of the archetype, giving the archetype grounding function $\mathcal{G}_m$, by taking the intersection of g over all definitions.

$$\mathcal{G}_m(p) = \bigcap_{g \in G} g(p)$$

where $G = \{g \mid g \text{ is the grounding function of some definition of } m\}$.

For any set of archetype definitions the corresponding set of archetype grounding functions can be determined by standard closure techniques.

5 Some Examples

Sizeable applications of the archetype mechanism can be found in chapters eight and nine. In this section the programme counter convention introduced in chapter two will be shown to be closely related to the archetype mechanism, and will be defined in terms of it. This opens the door to extensions of the convention, and two of these are introduced.

5.1 Hiding the Programme Counter

It should now be clear that an update scheme in which the programme counter has been "hidden" on both the left and right hand sides is in fact the body of an archetype definition. The convention is then equivalent to prefacing each such scheme with, for example, '$pc() =$', where pc is some archetype name not used elsewhere in the update plan, and adding the update scheme

$$\texttt{PC[pc] pc[}pc()\texttt{]qc} \Longrightarrow \texttt{PC[pc}'\texttt{] pc}'\texttt{[}pc()\texttt{]qc}.$$

to the update plan.

Indeed, the syntactic sugar in section 2 ensures that, if every update scheme in an update plan is written as a command, it suffices to prefix the plan with

'PC[pc] pc[$pc()$]qc$\Longrightarrow$PC[pc$'$] pc$'$[$pc()$]qc.
$pc() =$ '

5.2 Asynchronous Parallel Processors

It is also possible to define more than one programme counter, thus specifying asynchronous parallel processors with shared memory and identical instruction sets.

$$\begin{aligned}&\mathtt{PC_1[pc]\ pc[}pc()\mathtt{]qc} \Longrightarrow \mathtt{PC_1[pc']\ pc'[}pc()\mathtt{]qc}.\\ &\qquad\qquad\qquad\vdots\\ &\mathtt{PC_n[pc]\ pc[}pc()\mathtt{]qc} \Longrightarrow \mathtt{PC_n[pc']\ pc'[}pc()\mathtt{]qc}.\end{aligned}$$

5.3 Specifying the Instruction Cycle

Another possibility is "hiding" some other structure. A lower level specification, for example of the instruction cycle, might be more legible with the clock hidden.

$$\mathtt{CLOCK[}tick()\mathtt{]} \Longrightarrow \mathtt{CLOCK[}tick()\mathtt{]}.$$

For a simplified PDP-11 like machine the instruction cycle can be considered to consist of four steps; fetching the instruction (IFETCH), fetching the source operand (SRC), fetching the destination operand and its address (DST), and performing the operation and writing the result to the destination (EXEC). These steps can then be specified by:

$$\begin{aligned}&tick()\\ &\quad = \quad \mathtt{IFETCH\ PC[pc]\ pc[command]qc} \Longrightarrow \mathtt{SRC\ IR[command]\ PC[qc]}.\\ &\qquad\qquad\qquad\qquad\qquad\qquad\vdots\\ &\quad = \quad \mathtt{SRC\ IR + S[IMM]\ PC[pc]\ pc[x]qc} \Longrightarrow \mathtt{DST\ RX[x]\ PC[qc]}.\\ &\qquad\qquad\qquad\qquad\qquad\qquad\vdots\\ &\quad = \quad \mathtt{DST\ IR + D[REG\ r]\ R + r[y]} \Longrightarrow \mathtt{EXEC\ MAR[R + r]\ RY[y]}.\\ &\qquad\qquad\qquad\qquad\qquad\qquad\vdots\\ &\quad = \quad \mathtt{EXEC\ IR[ADD]\ RX[x]\ RY[y]\ MAR[dst]} \Longrightarrow \mathtt{IFETCH\ dst[x + y]}.\\ &\ \ \vdots\end{aligned}$$

where IR is the instruction register, MAR is the memory address register, RX and RY are the right and left port of the ALU respectively, and S and D are the offsets of the source and destination addresses in the command word. The specification assumes that the registers occupy a contiguous section of memory, with R as the left locator of register 0. A more elegant formulation of the same specification would be:

Addressing Modes.

$$\begin{aligned}&\mathtt{op(R + r, v)} = \mathtt{REG\ r\ R + r[v]}.\\ &\mathtt{op(pc, v)} \quad = \mathtt{IMM\ PC[pc]\ pc[v]qc} \Longrightarrow \mathtt{PC[qc]}.\\ &\qquad\qquad\vdots\end{aligned}$$

Operands.

$$\begin{aligned}&\mathtt{src(val)} \quad = \mathtt{op(\_, val)}.\\ &\mathtt{dst(ea, val)} = \mathtt{op(ea, val)}.\end{aligned}$$

Instructions.

$$\begin{aligned}
\texttt{IFETCH}() &= \texttt{PC[pc]}\ \texttt{pc[command]qc} && \Longrightarrow \texttt{IR[command]}\ \texttt{PC[qc]}.\\
\texttt{SRC}() &= \texttt{IR} + \texttt{S[src(x)]} && \Longrightarrow \texttt{RX[x]}.\\
\texttt{DST}() &= \texttt{IR} + \texttt{D[dst(ea, y)]} && \Longrightarrow \texttt{RY[y]}\ \texttt{MAR[ea]}.\\
&\\
\texttt{EXEC}() &= \texttt{IR[ADD]}\ \texttt{RX[x]}\ \texttt{RY[y]}\ \texttt{MAR[dst]} && \Longrightarrow \texttt{dst[x} + \texttt{y]}.\\
&\vdots
\end{aligned}$$

Instruction Cycle.

$$\begin{aligned}
\mathit{tick}() &= \texttt{IFETCH}() && \Longrightarrow \texttt{SRC}.\\
&= \texttt{SRC}() && \Longrightarrow \texttt{DST}.\\
&= \texttt{DST}() && \Longrightarrow \texttt{EXEC}.\\
&= \texttt{EXEC}() && \Longrightarrow \texttt{IFETCH}.
\end{aligned}$$

The Clock.

$$\texttt{CLOCK}[\mathit{tick}()] \Longrightarrow \texttt{CLOCK}[\mathit{tick}()].$$

6 Conclusions

Update Plans have already been shown useful as a specification formalism for abstract machines. The addition of a macro-like mechanism not only makes them suited to the specification of concrete machines, it also opens the door to applications in the specification of asynchronous parallel processes. Another extension of Update Plans to cover synchronous parallelism is presented in chapter nine.

Although the mechanism presented here is specific to Update Plans, the basic principle of having macro definitions and calls reflect the structure of the formalism in which they are defined could almost certainly be fruitfully applied to other languages and formalisms with well defined structures — for example, as has been suggested [55], the method used to determine grounding sets for archetypes could probably be applied to the problem of determining critical positions in predicates of Extended Affix Grammars [24, 42, 53, 74].

Chapter Eight Demonstrably Correct Linearisation of Intermediate Code

Code generation often generates intermediate code, before register allocation, with a tree-like structure, in which subtrees represent intermediate results in a calculation. This code must then be linearised — to instructions on some concrete machine — by passing intermediate results via registers, or via a stack.

With Update Plans both the concrete machine and an abstract machine for the intermediate code can be specified, using the same formalism. A transformation from intermediate code to concrete machine code can then be given. Using the properties introduced in chapter five this transformation can be shown to preserve semantics — i.e. to provide a correct translation.

In this chapter two related machines are specified:

- a linear code machine — a PDP-11 style machine, not actually extant, but similar enough to concrete machines to be used as an example,
- a machine for a tree-structured language, the language being typical of intermediate code emitted by a compiler.

The machines specified are based on machines described by Giegerich [28]. Transformations are defined from tree code to linear code (register allocation) and it is shown that these transformations define a translation, in the sense of chapter 6.

The aim of this chapter is two-fold. Firstly, the machine specifications serve as extensive examples of applications of the archetype mechanism introduced in chapter seven. In particular the tree machine illustrated in figures 1 to 8 in appendix VI provides a sizeable example of archetype definitions and their expansion. Secondly, the chapter suggests a way in which formal methods may be applied to a problem — showing register allocation to be correct — which has not previously been readily amenable to such an approach.

Section 1 presents the linear machine, which is summarised in figures 1 and 2 of appendix VI, and section 2 the tree machine, summarised in figures 1 and 3 of appendix VI. The scheme and archetype numbering in these, and other, sections is provided for ease of reference, as in chapter four, and does not form part of the schemes and archetypes. The transformations from tree code to linear code are given in section 3 and the semantic equivalence to the original code of the resulting code is shown in section 4. A short summary, and some suggestions for more extensive applications are given in section 5.

The transformation is not wholly realistic, since the concrete machine is assumed always to have a sufficient number of registers. This restriction is addressed in section 5.

The methods presented here can be of importance in developing provably correct code generators.

1 The Linear Code Machine

The linear code machine has eight addressing modes, two data transfer commands, four arithmetic and six comparison operators, and five programme flow commands. At this level of specification the number of registers is considered to be infinite.

1.1 The Specification

Addressing Modes. The following archetypes all define two results, the effective address and the value. In all of these the locator contained in register `r`, here consistently represented by `b`, is called the *base value* of the addressing mode. The addressing modes and corresponding archetypes are:

register, `REG`$(\cdot,\cdot)$ `r`. Direct addressing, the value accessed is the value in the argument (register `r`) of the addressing mode.

register deferred, `REGDEF`$(\cdot,\cdot)$ `r`. The address of the value is in the register `r`.

base + displacement, `BDISP`$(\cdot,\cdot)$ `r d`. The first argument is a register `r` containing a base adress, the second argument is the displacement `d`.

base + displacement deferred, `BDISPDEF`$(\cdot,\cdot)$ `r d`. The address of the value accessed is computed in the same way as the value in base + displacement mode.

predecrement, `PREDEC`$(\cdot,\cdot)$ `r`. Similar to register deferred mode, except that the address in the register `r` is decreased before the value is accessed.

postincrement, `POSTINC`$(\cdot,\cdot)$ `r`. Again similar to register deferred. The address in `r` is increased after the value is accessed.

immediate, `IMM v`. The argument `v` is the value accessed.

label, LAB t. Identical to immediate addressing. The value t should be a label.

[1.1] REG(r,b) r = r[b].
REGDEF(b,v) r = r[b] b[v].
BDISP(b+d,v) r d = r[b] b+d[v].
BDISPDEF(a,v) r d = r[b] b+d[a] a[v].
PREDEC(a,v) r = r[b] a[v]b ⟹r[a].
POSTINC(b,v) r = r[b] b[v]c ⟹r[c].

These archetypes are then used in definitions of classes of addressing modes. These classes are

wop writeable operands

rop readable operands

trg targets of jump commands

adr effective address operands

In this specification PREDEC and POSTINC have already been reserved for stack operations (*push* and *pop*) by assigning PREDEC to the wop class, and POSTINC to rop. A more general specification would include both PREDEC and POSTINC in both classes.

[1.2] wop(a,v) = REG(a,v). rop(v) = REG(_,v).
= REGDEF(a,v). = REGDEF(_,v).
= PREDEC(a,v). = BDISP(_,v).
= BDISP(a,v). = BDISPDEF(_,v).
= BDISPDEF(a,v). = POSTINC(_,v).
= IMM v.
adr(a) = REGDEF(a,_).
= BDISP(a,_). trg(t) = adr(t).
= BDISPDEF(a,_). = LAB t.

Data Transfer Commands. The definitions are self explanatory, MOV moves a value to a destination, MEA an "effective address".

[1.3] MOV rop(x) wop(a,_)⟹a[x].
MEA adr(x) wop(a,_)⟹a[x].

Arithmetic Commands. It is assumed that the standard arithmetic operators are defined in the standard environment. The arithmetic commands are then declared as archetypes.

[1.4] arith(x,y,y+x) = ADD.
arith(x,y,y − x) = SUB.
arith(x,y,y ∗ x) = MUL.
arith(x,y,y/x) = DIV =[x ≠ 0]⇛ .

[1.5] arith(x,y,r) rop(x) wop(a,y)⟹a[r].

Comparison Commands. The mechanism for the comparison operators is almost identical.

[1.6] $\texttt{bool}(x, y, x < y) = \texttt{CLT}.$
$\texttt{bool}(x, y, x \leq y) = \texttt{CLE}.$
$\texttt{bool}(x, y, x = y) = \texttt{CEQ}.$
$\texttt{bool}(x, y, x \neq y) = \texttt{CNE}.$
$\texttt{bool}(x, y, x \geq y) = \texttt{CGE}.$
$\texttt{bool}(x, y, x > y) = \texttt{CGT}.$

A comparison operator returns its value through the condition code register CC, rather than through the address of its second argument (as an arithmetic operator does).

[1.7] $\texttt{bool}(x, y, r)\ \texttt{rop}_1(x)\ \texttt{rop}_2(y) \Longrightarrow \texttt{CC}[r].$

Programme Flow. Three of the programme flow commands (the simple "jump" commands) can also be compressed into one update scheme.

[1.8] $\texttt{jump}(cc) = \texttt{JT CC}[cc].$
$\texttt{jump}(\neg(cc)) = \texttt{JF CC}[cc].$
$\texttt{jump}(\texttt{TRUE}) = \texttt{JMP}.$

[1.9] $\texttt{jump}(\texttt{cond})\ \texttt{trg}(t) =[\ \texttt{cond}\ ]\Rightarrow \texttt{PC}[t].$
$\quad " \quad =[\ \neg(\texttt{cond})\ ]\Rightarrow .$

The remaining two commands, "jump to subroutine" and "return", must be specified individually. Since the jump to subroutine command must preserve the contents of the PC its specification cannot be given in command form. The PC must be explicitly included on the left hand side.

[1.10] $\texttt{PC}[pc]\ \texttt{pc}[\texttt{JSR trg}(t)]qc\ \texttt{SP}[tp] \Longrightarrow \texttt{PC}[t]\ \texttt{SP}[sp]\ \texttt{sp}[qc]tp.$

$\texttt{RET SP}[sp]\ \texttt{sp}[pc]tp \Longrightarrow \texttt{PC}[pc]\ \texttt{SP}[tp].$

The linear machine specification is summarised in figures 1 and 2 of appendix VI.

1.2 A Small Example

The following example illustrates these archetypes, and the archetype expansion mechanism. In this example the update scheme

$$\texttt{arith}(x, y, r)\ \texttt{rop}(x)\ \texttt{wop}(a, y) \Longrightarrow a[r].$$

will be expanded, step by step, to

[1.11] $\texttt{SUB}\ (\texttt{POSTINC}\ r_1)\ (\texttt{REGDEF}\ r_2)$
$\quad r_1[b_1]\ b_1[x]c_1\ r_2[b_2]\ b_2[y]$
$\Longrightarrow r_1[c_1] \quad b_2[y - x]$

The expansion is presented in three stages: textual expansion, resolution and substitution. First the text of the archetype definitions is substituted for their calls. Then the equations generated during this process (the

$\{1.5\}$
$=$ $\texttt{arith(x,y,r) rop(x) wop(a,y)}$
$\texttt{==>}$ $\texttt{a[r].}$
$\{1.4$ $\texttt{arith(x,y,y - x) = SUB.}\}$
$=$ $\texttt{SUB rop(x) wop(a,y)}$ $r = y - x$
$\texttt{==>}$ $\texttt{a[r].}$
$\{1.2$ $\texttt{rop}(v_1) = \texttt{POSTINC}(\_, v_1).\}$
$=$ $\texttt{SUB POSTINC}(\_, v_1)\ \texttt{wop(a,y)}$ $x = v_1$
$\texttt{==>}$ $\texttt{a[r].}$
$\{1.1$ $\texttt{POSTINC}(b_1, v_2)\ \texttt{r}_1 = \texttt{r}_1[b_1]\ \texttt{b}_1[v_2]\texttt{c}_1 \texttt{==>} \texttt{r}_1[\texttt{c}_1].\}$
$=$ $\texttt{SUB (POSTINC r}_1)\ \texttt{wop(a,y)}$ $\_ = b_1, v_1 = v_2$
$\texttt{r}_1[b_1]\ \texttt{b}_1[v_2]\texttt{c}_1$
$\texttt{==>r}_1[\texttt{c}_1]$ $\texttt{a[r].}$
$\{1.2$ $\texttt{wop}(a_3, v_3) = \texttt{REGDEF}(a_3, v_3).\}$
$=$ $\texttt{SUB (POSTINC r}_1)\ \texttt{REGDEF}(a_3, v_3)$ $a = a_3, y = v_3$
$\texttt{r}_1[b_1]\ \texttt{b}_1[v_2]\texttt{c}_1$
$\texttt{==>r}_1[\texttt{c}_1]$ $\texttt{a[r].}$
$\{1.1$ $\texttt{REGDEF}(b_4, v_4)\ \texttt{r}_2 = \texttt{r}_2[b_4]\ \texttt{b}_4[v_4].\}$
$=$ $\texttt{SUB (POSTINC r}_1)\ \ \texttt{(REGDEF r}_2)$ $a_3 = b_4, v_3 = v_4$
$\texttt{r}_1[b_1]\ \texttt{b}_1[v_2]\texttt{c}_1\ \texttt{r}_2[b_4]\ \texttt{b}_4[v_4]$
$\texttt{==>r}_1[\texttt{c}_1]$ $\texttt{a[r].}$

Figure 1: Archetype expansion in the linear machine.

resolution set) are resolved. Substitutions for variables in the textual expansion, defined by these resolved equations, then result in the update scheme above.

Textual expansion. The arithmetic command scheme 1.5 expands as shown in figure 1. The archetypes, and their index numbers are included in figure 1 for clarity. The equations at the right are the equations added at each step to the resolution set, which is initially empty. The final expansion is

$[1.12]$ $\texttt{SUB (POSTINC r}_1)\ \ \texttt{(REGDEF r}_2)$
$\texttt{r}_1[b_1]\ \texttt{b}_1[v_2]\texttt{c}_1\ \texttt{r}_2[b_4]\ \texttt{b}_4[v_4]$
$\texttt{==>r}_1[\texttt{c}_1]$ $\texttt{a[r].}$

and a substitution must be found for the non-ground variables `r` and `a`.

Resolution. The resolution set is $\{r = x - y,\ x = v_1,\ \_ = b_1,\ v_1 = v_2,\ a = a_3,\ y = v_3,\ a_3 = b_4,\ v_3 = v_4\}$. The expressions in the resolution set can then be resolved as shown in figure 2, to give substitutions for `r` and `a`. In this schematic representation of resolution, arrows ($\rightarrow$) represent grounding due to expressions in the update scheme, and equality ($=$) grounding via an equation from the resolution set. For example, if r_1 is ground, then b_1 is also ground, due to the locator expression $\texttt{r}_1[\texttt{b}_1]$, and if v_4 is ground then v_3 is ground, due to the grounding set equation $v_3 = v_4$. An asterisk ($*$) indicates an equality that will be used in the substitution. Note that,

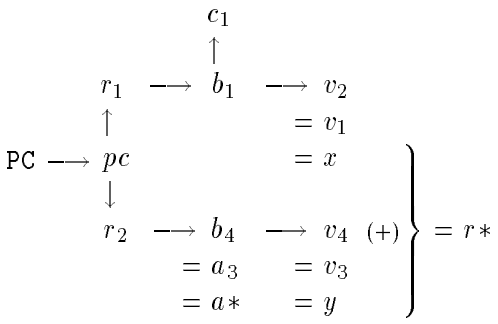
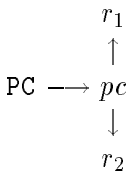


Figure 2: Resolving expressions in the linear machine.

for the sake of brevity, the initial part of the diagram in figure 2,



could have been omitted, as a similar part of diagram 8 on page VI.6 in appendix VI is.

Substitution. Substituting the values obtained in 1.12 gives

```
SUB (POSTINC r1)  (REGDEF r2)
    r1[b1] b1[v2]c1 r2[b4] b4[v4]
 ⟹r1[c1]          b4[v4 − v2].
```

which is obviously equivalent to scheme 1.11.

2 The Tree Machine

The machine in section 1 will now be extended. The result is a machine for a tree structured language. In this language operands have a tree like structure, in which subtrees represent intermediate results. The update schemes specifying this machine are identical to those of the linear machine. The specifications differ only in their archetype definitions. For example, the update scheme

[1.3] `MOV rop(x) wop(a,_)⟹a[x].`

taken from the linear machine, is also part of the specification of the tree machine. The archetype definitions permit the representation of intermediate results. One possible expansion of scheme 1.3 is

```
[2.1] MOV (IBDISP (IMOV (BDISP r d1) (IREG NIL)) d2)
          (IREGDEF (IMOV (IMM a) (IREG NIL)))
            r[b1] b1+d1[b2] b2+d2[v]⟹a[v].
```

(where brackets have been added to improve legibility). The constants `IBDISP` and `IREGDEF` are the "intermediate" versions of `BDISP` and `REGDEF`, respectively, taking as their base value the result of an intermediate

"command" such as IMOV. For example, the value of the first operand is fixed by first determining the value accessed by the innermost addressing mode BDISP r d_1. In specification 2.1 this is the value of b_2. The archetype definitions do not require a value to be accessed by IREG NIL. The value b_2 is then passed by the IMOV command to the outermost addressing mode (IBDISP (...) d_2), there to serve as base value.

Details of the expansion are given in figures 7 to 9 on pages VI.5 to VI.6 in appendix VI. The specification is given below, and summarised in figures 1 and 3 of appendix VI.

Although the full specification of the tree machine includes all of the update schemes and many of the archetypes from the specification of the linear machine they are repeated here for the sake of totality. Archetypes and update schemes unchanged from section 1 can be identified by their retention of the numbering given to them in that section. Constants which occur in both the linear machine and the tree machine, e.g. ADD and BDISP, are referred to as *linear constants*, and all others, e.g. IMOV and IREG, as *non-linear constants*. An expansion of a tree machine archetype is said to be *linear* if it contains no non-linear constants, and *non-linear* otherwise.

Addressing Modes. Intermediate addressing modes are added to the addressing mode archetypes. As in 1.1 the locator b in all of these archetype definitions is the base value of the addressing mode.

[1.1] REG(r, b) r = r[b].
REGDEF(b, v) r = r[b] b[v].
BDISP(b+d, v) r d = r[b] b+d[v].
BDISPDEF(a, v) r d = r[b] b+d[a] a[v].
PREDEC(a, v) r = r[b] a[v]b $\Longrightarrow$r[a].
POSTINC(b, v) r = r[b] b[v]c $\Longrightarrow$r[c].

[2.2] IREG(_, b) ir(b) = .
IREGDEF(b, v) ir(b) = b[v].
IBDISP(b+d, v) ir(b) d = b+d[v].
IBDISPDEF(a, v) ir(b) d = b+d[a] a[v].

The archetype ir(·) used here is defined on page 78.

The classes of addressing modes in 1.2 must now be redefined, and a new class, tmp(·), added. In fact two new classes are added, but the class kern(·, ·) is merely added for convenience, making the definitions of the other classes shorter.

[2.3] $\texttt{kern}(\texttt{a},\texttt{v}) = \texttt{REGDEF}(\texttt{a},\texttt{v}).$
$= \texttt{BDISP}(\texttt{a},\texttt{v}).$
$= \texttt{BDISPDEF}(\texttt{a},\texttt{v}).$
$= \texttt{IBDISP}(\texttt{a},\texttt{v}).$
$= \texttt{IBDISPDEF}(\texttt{a},\texttt{v}).$

$\texttt{rop}(\texttt{v}) = \texttt{kern}(\_,\texttt{v}).$
$= \texttt{IMM v}.$
$= \texttt{REG}(\_,\texttt{v}).$
$= \texttt{POSTINC}(\_,\texttt{v}).$
$= \texttt{IREG}(\_,\texttt{v}).$
$= \texttt{IREGDEF}(\_,\texttt{v}).$

$\texttt{wop}(\texttt{a},\texttt{v}) = \texttt{kern}(\texttt{a},\texttt{v}).$
$= \texttt{REG}(\texttt{a},\texttt{v}).$
$= \texttt{PREDEC}(\texttt{a},\texttt{v}).$
$= \texttt{IREGDEF}(\texttt{a},\texttt{v}).$

$\texttt{adr}(\texttt{a}) = \texttt{kern}(\texttt{a},\_).$
$\texttt{trg}(\texttt{t}) = \texttt{LAB t}.$
$\texttt{tmp}(\texttt{t}) = \texttt{IREG}(\_,\texttt{t}).$

The $\texttt{trg}(\cdot)$ class, which appears as the operand of the "jump" commands is restricted to include only (symbolic) labels. This will simplify the proof in section 4.2. A suggestion for relaxing this restriction is given in section 4.3.

Intermediate Results. The machine is extended with "intermediate" versions of the arithmetic and data transfer commands.

[2.4] $\texttt{iarith}(\texttt{y}+\texttt{x}) = \texttt{IADD rop}(\texttt{x})\ \texttt{tmp}(\texttt{y}).$
$\texttt{iarith}(\texttt{y}-\texttt{x}) = \texttt{ISUB rop}(\texttt{x})\ \texttt{tmp}(\texttt{y}).$
$\texttt{iarith}(\texttt{y}*\texttt{x}) = \texttt{IMUL rop}(\texttt{x})\ \texttt{tmp}(\texttt{y}).$
$\texttt{iarith}(\texttt{y}/\texttt{x}) = \texttt{IDIV rop}(\texttt{x})\ \texttt{tmp}(\texttt{y}) =[\ \texttt{x} \neq \texttt{0}\ ]\Rightarrow .$

[2.5] $\texttt{imov}(\texttt{x}) = \texttt{IMOV rop}(\texttt{x})\ (\texttt{IREG NIL}).$
$\texttt{imov}(\texttt{a}) = \texttt{IMEA adr}(\texttt{a})\ (\texttt{IREG NIL}).$

The second operand of IMOV and IMEA is actually superfluous, but makes the definition, given in section 3, of the transformation of tree code to linear code simpler. The addressing mode IREG NIL is *not* an expansion of the $\texttt{IREG}(\cdot,\cdot)$ archetype, and does not occur anywhere other than here. Its rôle in tree code is as a "dummy", a placeholder for the temporary register that may be assigned for an intermediate result in the corresponding linear code.

The intermediate arithmetic and data transfer commands can now be combined to form the class of intermediate results, which was used in the definition of intermediate addressing modes in 2.2.

[2.6] $\texttt{ir}(\texttt{r}) = \texttt{iarith}(\texttt{r}).$
$= \texttt{imov}(\texttt{r}).$

The remainder of the archetype definitions and update schemes are inherited from the linear machine.

Data Transfer Commands.

[1.3] $\texttt{MOV rop}(\texttt{x})\ \texttt{wop}(\texttt{a},\_) \Longrightarrow \texttt{a}[\texttt{x}].$
$\texttt{MEA adr}(\texttt{x})\ \texttt{wop}(\texttt{a},\_) \Longrightarrow \texttt{a}[\texttt{x}].$

Arithmetic Commands.

[1.4] $\texttt{arith}(\texttt{x},\texttt{y},\texttt{y+x}) = \texttt{ADD}.$
$\texttt{arith}(\texttt{x},\texttt{y},\texttt{y}-\texttt{x}) = \texttt{SUB}.$
$\texttt{arith}(\texttt{x},\texttt{y},\texttt{y}*\texttt{x}) = \texttt{MUL}.$
$\texttt{arith}(\texttt{x},\texttt{y},\texttt{y/x}) = \texttt{DIV} =[\ \texttt{x} \neq 0\ ]\Rightarrow .$

[1.5] $\texttt{arith}(\texttt{x},\texttt{y},\texttt{r})\ \texttt{rop}(\texttt{x})\ \texttt{wop}(\texttt{a},\texttt{y}) \Longrightarrow \texttt{a}[\texttt{r}].$

Comparison Commands.

[1.6] $\texttt{bool}(\texttt{x},\texttt{y},\texttt{x} < \texttt{y}) = \texttt{CLT}.$
$\texttt{bool}(\texttt{x},\texttt{y},\texttt{x} \leq \texttt{y}) = \texttt{CLE}.$
$\texttt{bool}(\texttt{x},\texttt{y},\texttt{x} = \texttt{y}) = \texttt{CEQ}.$
$\texttt{bool}(\texttt{x},\texttt{y},\texttt{x} \neq \texttt{y}) = \texttt{CNE}.$
$\texttt{bool}(\texttt{x},\texttt{y},\texttt{x} \geq \texttt{y}) = \texttt{CGE}.$
$\texttt{bool}(\texttt{x},\texttt{y},\texttt{x} > \texttt{y}) = \texttt{CGT}.$

[1.7] $\texttt{bool}(\texttt{x},\texttt{y},\texttt{r})\ \texttt{rop}_1(\texttt{x})\ \texttt{rop}_2(\texttt{y}) \Longrightarrow \texttt{CC}[\texttt{r}].$

Programme Flow.

[1.8] $\texttt{jump}(\texttt{cc}) = \texttt{JT}\ \texttt{CC}[\texttt{cc}].$
$\texttt{jump}(\neg(\texttt{cc})) = \texttt{JF}\ \texttt{CC}[\texttt{cc}].$
$\texttt{jump}(\texttt{TRUE}) = \texttt{JMP}.$

[1.9] $\texttt{jump}(\texttt{cond})\ \texttt{trg}(\texttt{t}) =[\ \texttt{cond}\ ]\Rightarrow\ \texttt{PC}[\texttt{t}].$
$\texttt{"} =[\ \neg(\texttt{cond})\ ]\Rightarrow .$

[1.10] $\texttt{PC}[\texttt{pc}]\ \texttt{pc}[\texttt{JSR trg(t)}]\texttt{qc}\ \texttt{SP}[\texttt{tp}] \Longrightarrow \texttt{PC}[\texttt{t}]\ \texttt{SP}[\texttt{sp}]\ \texttt{sp}[\texttt{qc}]\texttt{tp}.$

$\texttt{RET SP}[\texttt{sp}]\ \texttt{sp}[\texttt{pc}]\texttt{tp} \Longrightarrow \texttt{PC}[\texttt{pc}]\ \texttt{SP}[\texttt{tp}].$

The schemes and archetypes given above are collected in figures 1 and 3 of appendix VI. An example of expansion of some of these archetypes can be found in figures 7 to 9 on pages VI.5 to VI.6 in appendix VI.

3 Linearising Tree Code

Intermediate code is linearised by passing intermediate results via registers. The registers are realised as cells in a new store, **Aux**. In the following these auxiliary registers will be indicated by natural numbers (n, ...). This is a shorthand notation for $\texttt{AUX} + n$, ..., where **AUX** is some fixed locator in **Aux**.

A transformation function $\mathcal{T}[\![\cdot]\!]$ can be defined from tree code to linear code i.e. between the expansions of the left hand sides of archetype definitions and update schemes of the tree and linear machines. In order to facilitate the definition of $\mathcal{T}[\![\cdot]\!]$, as well as the proof that this defines a correct translation, the grammar in figure 3, which is based on the archetype grammar of the tree machine, is introduced, which defines (a superset of) the language of tree code commands. Trees *and* subtrees (intermediate results) are generated by the nonterminal ⟨tree⟩. The

⟨tree⟩ → ⟨dyadic⟩ ⟨op⟩ ⟨op⟩ |
⟨monadic⟩ ⟨op⟩ |
⟨niladic⟩ |
NIL

⟨dyadic⟩ → ⟨arith⟩ | ⟨iarith⟩ |
⟨move⟩ | ⟨imove⟩ |
⟨bool⟩

⟨monadic⟩ → ⟨jump⟩

⟨niladic⟩ → RET

⟨arith⟩ → ADD SUB MUL DIV
⟨iarith⟩ → IADD ISUB IMUL IDIV
⟨move⟩ → MOV MEA
⟨imove⟩ → IMOV IMEA
⟨bool⟩ → CGT CGEQ CEQ CNE CLEQ CLT

⟨jump⟩ → JT | JF | JMP | JSR

⟨op⟩ → ⟨dop⟩ | ⟨iop⟩

⟨dop⟩ → ⟨dmode$_1$⟩ ⟨x⟩ | ⟨dmode$_2$⟩ ⟨r⟩ ⟨d⟩

⟨dmode$_1$⟩ → REG REGDEF PREDEC POSTINC LAB IMM
⟨dmode$_2$⟩ → BDISP BDISPDEF

⟨iop⟩ → ⟨imode$_1$⟩ ⟨tree⟩ | ⟨imode$_2$⟩ ⟨tree⟩ ⟨d⟩

⟨imode$_1$⟩ → IREG IREGDEF
⟨imode$_2$⟩ → IBDISP IBDISPDEF

Figure 3: A grammar for tree code.

terminals ⟨x⟩, ⟨r⟩ and ⟨d⟩ are not specified in this grammar. Though the definition of $\mathcal{T}[\![\cdot]\!]$ is based on the grammar in figure 3, the update schemes and archetype definitions may impose further restrictions on configurations, and this will be indicated as necessary. Reference will also be made to the archetype definitions and update schemes where this is necessary to restrict the domain under consideration.

In the following definitions the prime symbol ′ is the postfix operator which translates all tree machine constants to their linear machine counterparts, i.e. MOV′ = MOV, IADD′ = ADD, IMM′ = IMM, IBDISP′ = BDISP, etc.

$\mathcal{T}[\![\cdot]\!]$ is defined in terms of $\mathcal{IR}[\![\cdot]\!]$ (defined on page 81), and $\mathcal{M}[\![\cdot]\!]$ (on

page 81).

$$\begin{aligned}
\mathcal{T}[\![dyadic\ op_1\ op_2]\!]\ n &= \mathcal{IR}[\![op_2]\!]\ n \\
&\quad\ \mathcal{IR}[\![op_1]\!]\ n+1 \\
&\quad\ dyadic'\ (\mathcal{M}[\![op_1]\!]\ n+1)\ (\mathcal{M}[\![op_2]\!]\ n) \\
\mathcal{T}[\![monadic\ op]\!]\ n &= \mathcal{IR}[\![op]\!]\ n \\
&\quad\ monadic'\ (\mathcal{M}[\![op]\!]\ n) \\
\mathcal{T}[\![niladic]\!]\ n &= niladic' \\
\mathcal{T}[\![\texttt{NIL}]\!]\ n &= \epsilon
\end{aligned}$$

Of the auxiliary functions $\mathcal{M}[\![\cdot]\!]$ is the simpler. It gives the linear machine addressing mode corresponding to its (tree machine) argument.

$$\begin{aligned}
\mathcal{M}[\![dmode_1\ x]\!]\ n &= dmode_1\ x \\
\mathcal{M}[\![dmode_2\ r\ d]\!]\ n &= dmode_2\ r\ d \\
\mathcal{M}[\![imode_1\ ir]\!]\ n &= imode'_1\ n \\
\mathcal{M}[\![imode_2\ ir\ d]\!]\ n &= imode'_2\ n\ d
\end{aligned}$$

The function $\mathcal{IR}[\![\cdot]\!]$ distinguishes between linear and non-linear addressing modes, returning code to compute an intermediate result, if necessary, by calling $\mathcal{T}[\![\cdot]\!]$.

$$\begin{aligned}
\mathcal{IR}[\![dmode_1\ x]\!]\ n &= \epsilon \\
\mathcal{IR}[\![dmode_2\ r\ d]\!]\ n &= \epsilon \\
\mathcal{IR}[\![imode_1\ ir]\!]\ n &= \mathcal{T}[\![ir]\!]\ n \\
\mathcal{IR}[\![imode_2\ ir\ d]\!]\ n &= \mathcal{T}[\![ir]\!]\ n
\end{aligned}$$

Note that $\mathcal{M}[\![op]\!]\ n$ and $\mathcal{IR}[\![op]\!]\ n$ are complementary in that $\mathcal{IR}[\![\cdot]\!]$ produces code which "computes", if necessary, the base value of *op* and places it in a register for the use of $\mathcal{M}[\![op]\!]$, which applies the required indirections and displacements to provide the effective address (if any) and value of *op*.

The main transformation function $\mathcal{T}[\![\cdot]\!]$ can be seen as a specification of a (simple) register allocation algorithm for a machine with an unlimited supply of registers. The brackets included, to increase legibility, in the `MOV` instruction on page 76 are essential for correct parsing of the argument of $\mathcal{T}[\![\cdot]\!]$. Correct bracketing will not be explicitly detailed here, but will be considered to be obvious.

4 Proving Semantic Equivalence

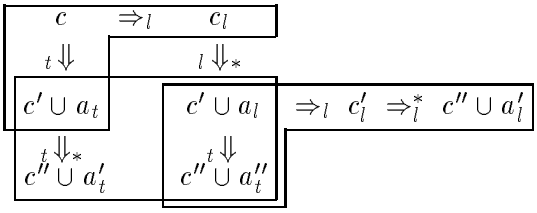
In addition to $\mathcal{IR}[\![\cdot]\!]$ it is assumed that, for any given configuration, there is a function from locators in the tree code programme store to locators in the linear code programme store, such that symbolic labels are preserved. I.e. if $\mathcal{L}[\![\cdot]\!]$ is this function, and `l[c]r` is part of the (tree code) programme store, then

$$\mathcal{L}[\![\texttt{l}]\!][\mathcal{T}[\![\texttt{c}]\!]]\mathcal{L}[\![\texttt{r}]\!]$$

will be part of the (linear code) programme store derived by applying $\mathcal{T}[\![\cdot]\!]$ and $\mathcal{L}[\![\cdot]\!]$. Note that $\mathcal{T}[\![\cdot]\!]$ is defined for individual tree machine commands. When applying it to a tree code programme store it will be necessary to map it across the sequence of instructions in the programme store.

It is claimed that $\mathcal{L}[\![\cdot]\!]$ and $\mathcal{T}[\![\cdot]\!]$, applied to the programme store and the contents of PC, when combined with the identity function for all other components of the configuration, form a correct translation, partial outside Aux, from configurations of the tree machines to configurations of the linear machine.

The proof is as sketched in chapter six. It will be shown that $\mathcal{T}[\![\cdot]\!]$ defines a local translation, partial outside Aux. Since Aux is trivially irrelevant, both locally and transitively, to the tree machine (it is a no-access memory), transitive translation can be established by inductive diagram chasing similar to that in section 5.1 of chapter six:



where $\Rightarrow_t$ is an update in the tree machine, and $\Rightarrow_l$ an update in the linear machine, and a_t, a_l, a'_t, a'_l and a''_t are all subsets of Aux.

The proof will establish that $(c \Rightarrow_t c' \cup a_t) \Longrightarrow (c \Rightarrow_l^* c' \cup a_l)$. The remaining non-trivial part of the translation box diagram

$$(c \Rightarrow_l c_l) \Longrightarrow (\exists c' : c_l \Rightarrow_l^* c' \cup a_l \wedge c \Rightarrow_t c' \cup a_t)$$

follows automatically from the deterministic nature of the linear machine.

At certain stages in the proof it will be necessary to consider translation partial outside a subset of Aux. The subset will always consist of those cells in Aux to the right of the locator AUX + n, for some n. Such a partial translation will be indicated by subscripting the translation symbol with n — i.e. $\triangleright_n \equiv \triangleright^{\overline{\{\mathtt{AUX}+k|k>n\}}}$.

The nonterminals of the grammar in figure 3 will be used consistently throughout the proof — a variable '*iop*' represents an expansion of the nonterminal ⟨iop⟩. Where necessary subscripts will be used to distinguish objects of the same type, e.g. op_1 and op_2.

4.1 An Important Lemma

The key to the proof is the following lemma:

Lemma For any non-linear operand *op*

$$\mathcal{IR}[\![op]\!]\ n\ \triangleright_n \mathtt{MOV}\ (\mathtt{IMM}\ b)\ (\mathtt{REG}\ n)$$

where b is the base value of *op*. $\diamond$

If *op* is non-linear, i.e. an *iop*, it must, in accordance with the grammar, be one of

IREG *tree*
IREGDEF *tree*
IBDISP *tree d*
IBDISPDEF *tree d*

where, from archetype definitions 2.2, 2.6, 2.4 and 2.5, and 2.3, *tree* is one of

iarith dop_1 iop_2
iarith iop_1 iop_2
imove dop_1 IREG NIL
imove iop_1 IREG NIL.

The occurrences of iop_1 and iop_2 call for a proof by induction. The proof will only be given for the second expansion of *tree* (*iarith* iop_1 iop_2), the other choices — including the third, which is the induction base — being analogous. Even more specifically, of all 48 possible choices for *op* only

IREGDEF (IADD iop_1 iop_2)

will be considered.

The proof is simple. Firstly,

$$\begin{aligned}\mathcal{IR}[\![op]\!]\ n &= \mathcal{IR}[\![\texttt{IREGDEF (IADD } iop_1\ iop_2\texttt{)}]\!]\ n \\ &= \mathcal{T}[\![\texttt{IADD } iop_1\ iop_2]\!]\ n \\ &= \mathcal{IR}[\![iop_2]\!]\ n \\ &\quad\ \mathcal{IR}[\![iop_1]\!]\ n+1 \\ &\quad\ \texttt{ADD } (\mathcal{M}[\![iop_1]\!]\ n+1)\ (\mathcal{M}[\![iop_2]\!]\ n).\end{aligned}$$

Assuming the base values of iop_1 and iop_2 to be b_1 and b_2 respectively gives, applying the induction hypothesis,

$$\mathcal{IR}[\![iop_2]\!]\ n\ \triangleright_n\ \texttt{MOV (IMM } b_2\texttt{) (REG } n\texttt{)}$$

and

$$\mathcal{IR}[\![iop_1]\!]\ n+1\ \triangleright_{n+1}\ \texttt{MOV (IMM } b_1\texttt{) (REG } n+1\texttt{)}$$

The only cell covered by the $\triangleright_{n+1}$ relation and not by the $\triangleright_n$ relation is register $n+1$. The contents of this cell will be updated by the MOV (IMM b_1) (REG $n+1$) command (see below) and therefore

$$\begin{aligned}\mathcal{IR}[\![op]\!]\ n\ \triangleright_{n+1}\ &\texttt{MOV (IMM } b_2\texttt{) (REG } n\texttt{)} \\ &\texttt{MOV (IMM } b_1\texttt{) (REG } n+1\texttt{)} \\ &\texttt{ADD } (\mathcal{M}[\![iop_1]\!]\ n+1)\ (\mathcal{M}[\![iop_2]\!]\ n).\end{aligned}$$

It follows, from archetype definitions 2.4 and 2.3, that iop_2 can only be IREG $tree_2$, while, by the grammar, there are again four possible choices for iop_1. Again only one case will be considered, IBDISP $tree_1$ d_1, the other cases again being analogous. Then $\mathcal{M}[\![iop_1]\!]\ n+1$ is BDISP $n+1$ d_1, and $\mathcal{M}[\![iop_2]\!]\ n$ is REG n.

Now iop_1 and iop_2 are expansions of archetypes IBDISP(b_1+d_1, r_1) and REG$(_, b_2)$, and any update scheme covering *op* will contain, inter alia, the contexts of these (expanded) archetypes, and in particular, on the left hand side, the locator expression $b_1 + d_1[v_1]$.

Now any configuration satisfying

PC[pc] pc[MOV (IMM b_2) (REG n)]~
qc[MOV (IMM b_1) (REG $n+1$)]~
rc[ADD (BDISP $n+1$ d_1) (REG n)]sc $\quad b_1 + d_1[v_1]$

updates deterministically to one satisfying

$$\begin{array}{ll} \texttt{PC}[\texttt{rc}] & \texttt{rc}[\texttt{ADD}\ (\texttt{BDISP}\ n+1\ d_1)\ (\texttt{REG}\ n)]\texttt{sc} \\ \quad b_1 + d_1[v_1] & n+1[b_1] \qquad\qquad n[b_2] \end{array}$$

and then to one satisfying

$$\texttt{PC}[\texttt{sc}]\ n[v_1 + b_2].$$

Clearly, then,

$$\mathcal{IR}[\![op]\!]\ n\ \triangleright_n \texttt{MOV}\ (\texttt{IMM}\ v_1 + b_2)\ (\texttt{REG}\ n).$$

Furthermore, according to the definition of the tree machine, $v_1 + b_2$ is indeed the base value of

$$\texttt{IREGDEF}\ (\texttt{IADD}\ (\texttt{IBDISP}\ tree_1\ d_1)\ (\texttt{IREG}\ tree_2))$$

where $tree_1$ and $tree_2$ are expansions of $\texttt{ir}(b_1)$ and $\texttt{ir}(b_2)$ respectively. This establishes the lemma.

4.2 The Proof

Tree code is either of the form *dyadic op op*, or of the form *monadic op*, or of the form *niladic*. Examination of the update schemes suffices to show that $\mathcal{T}[\![niladic]\!]\ n\ \triangleright_n\ niladic$. The operand of a *monadic op* is restricted by the archetype definitions to a `LAB t`. This case reduces to the case for *niladic*, as does a *dyadic dop dop*. The most general case is *dyadic iop iop*. The proof that $\mathcal{T}[\![dyadic\ iop\ iop]\!]\ n\ \triangleright_n\ dyadic\ iop\ iop$ is very similar to the proof of the lemma in section 4.1, though it no longer requires induction, and will not be presented. The remaining two cases — i.e. a *dyadic op op* in which one of the operands is a *dop* — are, in structure at least, simpler. Once it has been established that $\mathcal{T}[\![niladic]\!]\ n\ \triangleright_n\ niladic$, that $\mathcal{T}[\![monadic\ op]\!]\ n\ \triangleright_n\ monadic\ op$ and that $\mathcal{T}[\![dyadic\ op\ op]\!]\ n\ \triangleright_n\ dyadic\ op\ op$ it is trivial to show that $\mathcal{T}[\![code]\!]\ 0\ \triangleright^{\overline{\texttt{Aux}}}\ code$. Care should be taken with the programme flow commands, *jump*. The archetypes require the operand of a *jump* to be an expansion of `trg(t)`. In the tree machine this must be a label — a locator in the programme store of the tree machine. Preservation of labels then ensures the required semantic equivalence. It is for this reason that `trg(t)` is restricted to expand to `LAB t` in the tree machine.

4.3 Possible Extensions

Stacks. The transformation given assumes a sufficient supply of auxiliary registers. An alternative definition could provide translations in which intermediate results are passed via a stack, if there are insufficient auxiliary registers. The transformation to code for passing results via the stack is simply presented here, without a proof of its correctness. This is, of course, not the only possible definition, nor even probably the most "preferable". It is however almost certainly the simplest. The proof is roughly along the same lines as that already presented.

$$\begin{array}{lll} \mathcal{T}[\![dyad\ op_1\ op_2]\!]\ n & = & \mathcal{IR}[\![op_1]\!]\ n \\ & & \texttt{MOV}\ (\mathcal{M}[\![op_1]\!]\ n)\ (\texttt{PREDEC SP}) \\ & & \mathcal{IR}[\![op_2]\!]\ n \\ & & dyad'\ (\texttt{POSTINC SP})\ (\mathcal{M}[\![op_2]\!]\ n) \end{array}$$

This transformation can either be added to the definition of $\mathcal{T}[\![\cdot]\!]$, to be called whenever the number of auxiliary registers available falls below a critical reserve, or used to make $\mathcal{T}[\![\cdot]\!]$ a relation specifying both code for passing intermediate results via auxiliary registers and code in which results are passed via the stack. In both cases, proving that the new transformation preserves the semantics suffices, in combination with the proof above, to ensure the correctness of the translation (or translations) provided by $\mathcal{T}[\![\cdot]\!]$.

Another possibility is to have $\mathcal{T}[\![\cdot]\!]$ return some error value when the auxiliary registers and/or the stack are exhausted. Correct translation must then be proved for any code for which $\mathcal{T}[\![\cdot]\!]$ does not return this error value.

Jumps. Restricting the targets of jumps, in particular subroutine jumps, to symbolic labels is fairly drastic, making it impossible, for example, to return procedures or functions as results — or, at least, to call a procedure via such a result. This restriction can be relaxed fairly simply.

Both the linear machine and the tree machine have a programme store, say `ProgStore`. Since `ProgStore` is a programme store any `t` in a `trg(t)` will be of type `ProgStore`. The property required of the target of a jump is not that it *be* a (symbolic) label, but that it *have the value* of such a label. Restricting the target of a jump to a `LAB t` corresponds to requiring that `t` be a label. The lesser restriction can be achieved by requiring all objects of type `ProgStore` in the initial configuration to be symbolic labels, and, furthermore, requiring that no object of type `ProgStore` in the update plan be any expression more complicated than a constant (a symbolic label), or a single variable. Implicitly, such objects may therefore not appear as an argument of, for example, an `ADD` or `IADD`. This ensures that labels may be copied, and moved from cell to cell, but that no "new" `ProgStore` objects can be created. In any update script satisfying these conditions almost any addressing mode may be used in the operand of a jump instruction.

5 Conclusions

Update Plans, and in particular the archetype mechanism, make it possible to reason formally about low level activities. Transformations at this level can be proven correct. While the proof outlined in this chapter is fairly baroque, it is hard to see how any possible proof could be simpler. The combinatorial explosion of addressing modes almost certainly makes extensive case analysis unavoidable. Update Plans, and in particular the archetype mechanism, at least offer the means to emphasise the similarities, rather than the differences, between the cases, and possibly open the way to (partial) automation of the proof obligation.

Register allocation is but one transformation. The methods presented here can also be applied to other such transformations, be it at tree code level (with a slight abuse of notation):

$$dyad\ (\texttt{IREG}\ (\texttt{IMOV}\ op_1(\texttt{IREGNIL})))\ op_2 \equiv dyad\ op_1\ op_2),$$

or at linear machine level where a peephole optimiser could be defined.

Since all these applications preserve semantics they can also be combined; first optimising at the tree-code level, then performing register allocation, and then optimalising the resultant linear code.

While nondeterminism has been mentioned in passing in chapters one and seven, and specified implicitly in chapter three, it has not yet been covered in detail. Implementational aspects of nondeterminism are covered in chapter ten. This chapter is about the relation of nondeterminism to synchronous parallelism.

In chapter seven nondeterminism in Update Plans was exploited for the specification of asynchronous parallel processors. An adaptation of nondeterminism can be used to specify synchronous parallelism. This adaptation is described in sections 1, 2 and 3. An application — pipelining in a partial specification of the Berkeley RISC II CPU — is presented in section 4.

1 Informal Introduction

Update Plans as defined in chapters two and three, and as used in chapters four and eight, intuitively "work" by instantiating all applicable update schemes and then making a nondeterministic choice which of the update rules thus obtained to apply. An alternative would be to apply all of the update rules simultaneously. This is the idea behind a *parallel block*. A parallel block is a set of update schemes all applicable instantiations of which will be applied at the same time, if possible. The caveat is that some of these instantiations may have conflicting right hand sides. If this is the case, then none of the schemes are applied, the reasoning being that the update rules are applied as if they all formed one update rule, and update rules with inconsistent right hand sides may not be applied, as detailed in chapter three.

2 Syntax

Parallel blocks are delimited by the *open parallel block symbol*, '(||', and the *close parallel block symbol*, '||)'. The pipeline symbol '||' may be used anywhere in an update plan that whitespace is permitted. This can be used to emphasise a parallel block by prefixing each line with a pipeline

symbol.

Example 1

The parallel block

```
(||
    IR+S[src(x)]    ==>RX[x].
    IR+D[dst(ea,y)]==>RY[y] MAR[ea].
||)
```

may be written

```
(||
 ||   IR+S[src(x)]    ==>RX[x].
 ||   IR+D[dst(ea,y)]==>RY[y] MAR[ea].
 ||)
```

or, making use of typesetting possibilities

```
(
 ‖  IR+S[src(x)]    ==>RX[x].
 ‖  IR+D[dst(ea,y)]==>RY[y] MAR[ea].
 ‖ )
```

In the last case, and under the condition that this is the only use of the pipeline symbol, the open and close parallel block symbols may be omitted, giving

```
‖ IR+S[src(x)]    ==>RX[x].
‖ IR+D[dst(ea,y)]==>RY[y] MAR[ea].
```

The grammar given in chapter two, and modified in chapters five and seven must again be extended. Add the production rules

⟨item⟩ → ⟨parallel block⟩

⟨parallel block⟩ → (|| ⟨alternatives⟩ ||)

The full grammar of Update Plans, with all extensions, is given in appendix II.

3 Formal Specification

In chapter three the semantics of Update Plans was defined in terms of applications of update schemes: the update relation $\Rightarrow_s$ is defined in terms of the function $\mathcal{S}$ (which given a left hand side, a guard and a configuration gives a set of substitutions under which an update scheme having that left hand side and guard will be applicable to the given configuration), and the interpretation function $\mathcal{I}[\![\cdot]\!]$ (which interprets the instantiation of a right hand side). In the definition of $\Rightarrow_s$ on page 22 the relation can be said to

create and apply applicable update rules "on the fly". An equivalent view is to first derive applicable update rules and to define the semantics of Update Plans in terms of update rule applications.

Given an update plan, P, and a configuration, c, the set of update rules from P applicable to c is given by

$$\mathcal{R}\ P\ c = \{(l^\sigma, r^\sigma) \mid \exists g : (l, g, r) \in P, \sigma \in \mathcal{S}\ l\ g\ c \text{ and } r^\sigma \text{ is consistent}\}.$$

This can be simplified to give the set of updates (right hand sides), since the left hand sides are only relevant for determining which instantiations of which update schemes are applicable.

$$\mathcal{U}\ P\ c = \{r \mid (l, r) \in \mathcal{R}\ P\ c\}.$$

The relation between configurations defined by an update plan P can now be expressed as

$$c \Rightarrow_P c' \equiv \exists r \in \mathcal{U}\ P\ c : r \text{ is consistent} \wedge c' = \mathcal{I}[\![r]\!] \uplus c.$$

This is equivalent to the definition of $\Rightarrow_P$ given in chapter three but gives, perhaps, a clearer picture of the possibilities from which a nondeterministic choice is made during evaluation of an update script.

The semantics of parallel blocks can be defined in the same way. Rather than making a nondeterministic choice between applicable update rules all such rules are applied simultaneously. In terms of $\mathcal{U}$, an update by parallel block B is defined by

$$c \Rightarrow_B c' \equiv \bigcup(\mathcal{U}\ B\ c) \text{ is consistent} \wedge c' = \mathcal{I}\left[\!\!\left[\bigcup(\mathcal{U}\ B\ c)\right]\!\!\right] \uplus c.$$

For the case that $\mathcal{U}\ B\ c$ is a singleton set this reduces to the definition of $\Rightarrow_s$ given in chapter three.

Example 2

If the set of applicable update rules from a parallel block consists of the update rules

```
IR+S[REG R0] R0[X]⟹RX[X].
```

and

```
IR+D[REG R1] R1[Y]⟹RY[Y] MAR[R1].
```

these will be applied simultaneously as if the update rule

```
IR+S[REG R0] R0[X] IR+D[REG R1] R1[Y]
  ⟹RX[X]  RY[Y] MAR[RY].
```

were to be applied.

In keeping with the macro character of archetypes, archetype expansion conceptually takes place before interpretation of parallel blocks, and any expansions remain within the parallel block.

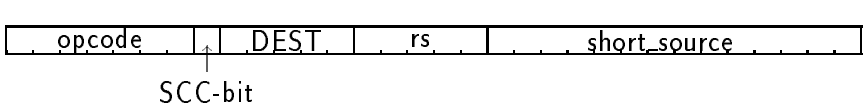


Figure 1: Format of a RISC II arithmetic instruction.

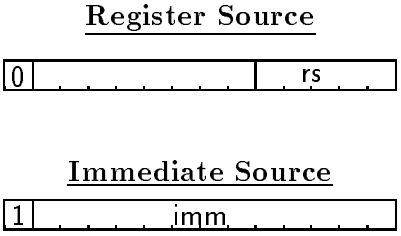


Figure 2: Short-source instruction field formats.

4 An Example: The Berkeley RISC II CPU

In the following example a subset of the Berkeley RISC II instruction set will be specified, first at the instruction level, to provide a general view; then at the instruction cycle level, as in chapter seven, but without pipelining; and then, finally, with pipelining, first without internal forwarding, and then with internal forwarding. Only a small subset of the instruction set is specified, in order to limit the length of this example. The specification of this subset will, however, be fairly detailed, and will also serve as an illustration of the use of Update Plans for the specification of concrete machines. In particular, the instruction level specification is fairly long. It is, however, complete, for this subset. The specification is based on the description of the Berkeley RISC II CPU given by Katavenis [35].

4.1 Instruction Set

The subset that will be specified is that of the arithmetic commands. An arithmetic instruction on the RISC II has a *short-immediate format*, as shown in figure 1 in which: DEST is the destination, which must be a register; rs, which must also be a register, is the first operand; and short_source is the second. The second operand, short_source, is a *short-immediate operand* which may have one of two formats. These formats are shown in figure 2.

Addressing Modes. The RISC II has three types of register — global registers shared by all routines, local registers addressable in a local window indicated by the current window pointer CWP, and register 0, which always has value 0. In the following definition register 0 will be considered a global register. It will be treated as a special case as the need arises.

```
reg(GBASE+r) = global(r).
reg(cwp+r)   = local(r) CWP[cwp].
```

GBASE is the left locator of register 0. The two archetypes global(r) and local(r) distinguish between global and local registers. The global registers are registers 0 to 9, while registers 10 to 31 are local. The archetypes are defined as:

$$\texttt{global(r)} = \texttt{r} \; =[\; 0 \leq \texttt{r} \leq 9 \;]\Rightarrow .$$
$$\texttt{local(r)} \; = \texttt{r} =[\; 10 \leq \texttt{r} \leq 31 \;]\Rightarrow .$$

Note that the variable r in these archetype definitions will create a new variable in any update scheme in which the archetype occurs, said variable then obtaining its value by instantiation in the usual way.

A register source can now be defined.

$$\texttt{rs(0)} \quad = \texttt{reg(GBASE)}.$$
$$\texttt{rs(val)} = \texttt{reg(ea)}\ \texttt{ea[val]} =[\; \texttt{ea} \neq \texttt{GBASE} \;]\Rightarrow .$$

The last instruction field of a short-immediate format instruction can have one of two formats, and this is specified by:

$$\texttt{short(val)} = \texttt{REG rs(val)}.$$
$$= \texttt{IMM val}.$$

Condition Codes. Condition codes are set if the SCC-bit is ON.

$$\texttt{scc(\_)} \quad = \texttt{OFF}.$$
$$\texttt{scc(flags)} = \texttt{ON}\Longrightarrow\texttt{CC[flags]}.$$

For arithmetic operations the values of the condition codes, in the order N, Z, V, C, are given by

$$\texttt{aflg(v)} = (\texttt{v} <_\texttt{s}) \; (\texttt{v} = 0) \; \neg(\texttt{MIN}_\texttt{s} \leq_\texttt{s} \texttt{v} \leq_\texttt{s} \texttt{MAX}_\texttt{s}) \; (\texttt{v} >_\texttt{u} \texttt{MAX}_\texttt{u}) \; |.$$

The operators $<_\texttt{s}$, $\leq_\texttt{s}$ (signed comparison operators), $>_\texttt{u}$ (unsigned comparison) and $=$ are assumed to be defined elsewhere, as are the constants $\texttt{MAX}_\texttt{s}$, $\texttt{MIN}_\texttt{s}$ and $\texttt{MAX}_\texttt{u}$. In a complete, bit level, specification, the values of the flags might be determined with reference to the values of the source operand, as well as that of the result. Such a specification will not be given here.

The SCC-bit in an arithmetic instruction can now be represented by scc(aflg(v)), where v is the result of the arithmetic operation. This will expand to either

$$\texttt{OFF}\Longrightarrow.$$

or

$$\texttt{ON}\Longrightarrow\texttt{CC}[(\texttt{v} <_\texttt{s}) \; (\texttt{v} = 0) \; \neg(\texttt{MIN}_\texttt{s} \leq_\texttt{s} \texttt{v} \leq_\texttt{s} \texttt{MAX}_\texttt{s}) \; (\texttt{v} >_\texttt{u} \texttt{MAX}_\texttt{u})].$$

Arithmetic Instructions. The arithmetic opcodes are defined analogously to those in chapter eight.

$$\texttt{arith(x,y,x+y)} = \texttt{ADD}.$$
$$\texttt{arith(x,y,x+y+c)} = \texttt{ADDC C[c]}.$$
$$\texttt{arith(x,y,x-y)} = \texttt{SUB}.$$
$$\texttt{arith(x,y,x-y-(1-c))} = \texttt{SUBC C[c]}.$$
$$\texttt{arith(x,y,y-x)} = \texttt{SUBI}.$$
$$\texttt{arith(x,y,y-x-(1-c))} = \texttt{SUBCI C[c]}.$$

```
ADD OFF r s (IMM y) CWP[cwp] cwp+s[x]
    =[ r ∈ GLBL ∧ r ≠ 0 ∧ s ∈ LOC ∧ cwp+s ≠ GBASE ]⇒
        GBASE+r[y+x].
```

Figure 3: A typical Berkeley RISC II arithmetic instruction. This instruction takes the value y, adds it to the contents of local register s, and places the result in register r. The condition codes are not set.

The register C contains the carry bit. All arithmetic instructions are now defined by

```
arith(x, y, r) scc(aflg(r)) reg(ea) rs(x) short(y)
      =[ ea = GBASE ]⇒ .
  "   =[ ea ≠ GBASE ]⇒ ea[r].
```

A possible concrete example of this update scheme, with the archetypes fully expanded, is given in figure 3. Some expressions derived during the expansion have been simplified.

4.2 The Instruction Cycle

The instruction cycle consists of three phases; instruction fetch, execute, and write. In the instruction fetch phase the instruction currently addressed by the PC is copied to the instruction register IR. The execute phase accesses the operands and performs the operation, setting the condition codes, if necessary, and placing the result in the RES register. In order to keep all accesses of the internal structure of an instruction within one phase, the destination address is also copied to the DST register, where it will be used by the write phase. The execute phase also updates the PC to contain the address of the next instruction to be executed. The PC is only updated at execution, rather than when the instruction is fetched, since addressing may be relative to the PC for some instructions. Finally, the write phase copies the result from the RES register to the destination. As in chapter seven, a clock is introduced to ensure that the phases take place sequentially. In the next step, pipelining, the clock will be eliminated.

```
FETCH() =
  PC[pc] pc[instruction]⟹IR[instruction].
EXEC() =
  IR[arith(x, y, r) scc(aflg(r)) reg(dst) rs(x) short(y)] PC[pc]
      ⟹RES[r] DST[dst] PC[pc+WORD].
WRITE()
  =  DST[dst] RES[r]=[ dst = GBASE ]⇒.
  =        "       =[ dst ≠ GBASE ]⇒dst[r].
```

WORD is the length of an instruction. The instruction cycle is defined by:

```
tick() = FETCH() ⟹ EXEC.
       = EXEC()  ⟹ WRITE.
       = WRITE() ⟹ FETCH.
```

The update scheme is

$$\texttt{CLOCK}[\textit{tick}()] \Longrightarrow \texttt{CLOCK}[\textit{tick}()].$$

Note that this makes use of the mechanism, discussed in chapter seven, which matches archetype calls on left and right hand sides. It is easy to establish that the instruction cycle specification is a correct translation of the higher level specification.

4.3 Pipelining

The first step in introducing pipelining is eliminating the clock, and introducing a parallel block in order to synchronise the phases of the instruction cycle. The *tick*() archetype bodies are elevated to update schemes within the parallel block.

$$\begin{array}{l}\texttt{PC[pc] pc[instruction]} \Longrightarrow \texttt{IR[instruction]}. \\ \texttt{IR[arith(x,y,r) scc(aflg(r)) reg(dst) rs(x) short(y)] PC[pc]} \\ \qquad \Longrightarrow \texttt{RES[r] DST[dst] PC[pc+WORD]}. \\ \texttt{DST[dst] RES[v]} =[\ \texttt{dst} = \texttt{GBASE}\ ] \Rightarrow . \\ \qquad\quad " \qquad =[\ \texttt{dst} \neq \texttt{GBASE}\ ] \Rightarrow \texttt{dst[v]}.\end{array}$$

For most configurations this specification will have the same semantics as the specification of the non-pipelined machine in section 4.2. For example, given initial configuration

$$\begin{array}{ll}[4.1] & \texttt{pc}_1\texttt{[ADD OFF 1 2 (IMM 1)]}\sim \\ & \texttt{pc}_2\texttt{[ADD OFF 3 4 (IMM 1)]}\sim \\ & \texttt{pc}_3\texttt{[ADD OFF 0 0 (REG 0)]}\sim \\ & \texttt{pc}_4\texttt{[ADD OFF 0 0 (REG 0)]pc}_5 \\ \\ & \texttt{PC[pc}_1\texttt{] IR[ADD 0 0 (REG 0)]} \\ & \texttt{RES[0] DST[GBASE] GBASE[0 1 2 ...]} \\ & \texttt{CLOCK[FETCH]}.\end{array}$$

both the specification in this section, and that in section 4.2, will have the net effect

$$\texttt{PC[pc}_1\texttt{]} \Longrightarrow \texttt{PC[pc}_5\texttt{] GBASE+1[3] GBASE+3[5]}.$$

In 4.1 the locator expression GBASE[0 1 2 ...] specifies that register 0 contains zero, register 1, one, etc. The expression DST[GBASE] initialises the destination to REG 0, ensuring no unwanted write access will take place before the pipeline is full. The last two instructions are included as null operations, giving the pipeline time to clear.

However not all configurations are as "well behaved" as 4.1. In the pipelined machine care must be taken if an instruction uses the result of the previous instruction. Since the operands of the next instruction are determined before the result of the previous instruction has been written to its destination, a stale value may be used. For example, if the second instruction in 4.1 were to be ADD OFF 3 1 (IMM 1) rather than ADD OFF 3 4 (IMM 1) the net effect of the sequential specification would be

$$\texttt{PC[pc}_1\texttt{]} \Longrightarrow \texttt{PC[pc}_5\texttt{] GBASE+1[3] GBASE+3[4]}.$$

while that of the pipelined specification would be still be

$$\mathtt{PC}[\mathtt{pc}_1] \Longrightarrow \mathtt{PC}[\mathtt{pc}_5]\ \mathtt{GBASE}{+}1[3]\ \mathtt{GBASE}{+}3[5].$$

This problem is avoided by using *internal forwarding*. The execution phase must check if the result waiting to be written should be used instead of a stale value. This can be done by revising the specification of a register source.

$$\begin{aligned}
\mathtt{rs}(0) &= \mathtt{reg}(\mathtt{GBASE}).\\
\mathtt{rs}(\mathtt{res}) &= \mathtt{reg}(\mathtt{dst})\ \mathtt{DST}[\mathtt{dst}]\ \mathtt{RES}[\mathtt{res}] =[\ \mathtt{dst} \neq \mathtt{GBASE}\ ]\Rightarrow .\\
\mathtt{rs}(\mathtt{val}) &= \mathtt{reg}(\mathtt{ea})\ \mathtt{DST}[\mathtt{dst}]\ \mathtt{ea}[\mathtt{val}] =[\ \mathtt{ea} \neq \mathtt{GBASE} \wedge \mathtt{ea} \neq \mathtt{dst}\ ]\Rightarrow .
\end{aligned}$$

The definition of $\mathtt{rs}(\cdot)$ has been extended. The second declaration "intercepts" a result of the previous instruction. DST and RES contain the result register and value of the previous instruction, which are still in the pipeline. If $\mathtt{rs}(\cdot)$ requires the value in that result register it should take the "new" value from RES, rather than the "old" value at ea, since this will not yet have been updated.

The rest of the specification is unchanged. The techniques of chapter eight could be applied to these specifications in order to prove their semantic equivalence. Certain restrictions will have to be imposed on the code, in particular on instructions immediately after a jump instruction, in order to make this possible.

5 Conclusions

The combination of the archetype mechanism and a simple notation for indicating parallelism makes it possible to write short, concise specifications of pipelining architectures. Using methods similar to those applied in chapter eight it would be possible to derive clearly defined conditions for (generated) code which would guarantee equivalence of pipelined and non-pipelined code. It also becomes possible to derive provably correct optimalisers which take advantage of pipelining.

Chapter Ten Implementational Aspects

The previous chapters of this thesis have defined Update Plans. In this chapter a possible implementation of the specification language will be presented. It is not the intention to provide a definitive specification of an implementation, but rather to illustrate techniques that could be used for such an implementation.

Section 1 introduces certain notational conventions specific to this chapter. Sections 2 and 3 discuss the implementation of deterministic Update Plans, section 2 for basic Update Plans as defined in chapters one, two and three, and section 3 for Update Plans with archetypes as defined in chapter seven. Section 4 discusses how backtracking could be implemented for nondeterministic Update Plans, and how this mechanism could be adapted to implement the parallel blocks from chapter nine. Finally, section 5 evaluates the implementation described, and suggests some techniques for improving its efficiency.

The key to the implementation is access to the (internal representations of) values of terms appearing in an update scheme. The implementation presented takes a straightforward approach, storing values, or pointers to values, in a *value store*. A more realistic approach would store some values in temporary registers. This is, however, irrelevant to the further implementation, as long as an access path to the values can be statically determined. The implementation first constructs a storage structure for values, then checks the guard for applicability and the right hand side for consistency and then, if these checks are successful, applies the update scheme.

1 Notes on Notation

This chapter applies certain notational conventions. These are summarised here.

Canonical Forms. The update plan to be implemented will be assumed to be in canonical form — i.e. all syntactic sugar will have been removed and all guards will be made explicit.

Update schemes may contain implicit guards, either due to the requirements of consistent substitution, as in

$$\texttt{R1[x] R2[x]} \ \ldots \Longrightarrow \ldots \ .$$

in which the contents of registers `R1` and `R2` must be identical; or by the occurrence of (semi-)ground terms in the scheme, as in

$$\texttt{R0[2]} \ \ldots \Longrightarrow \ldots \ .$$

in which the contents of `R0` must be the constant `2`.

The grounding analysis algorithm presented in chapter five will detect such implicit guards. An update scheme which has been desugared, and in which in which all implicit guards have been made explicit, is said to be in *canonical form*.

Example 1 (a)

For example, update scheme 2.1 of the FLIP machine in chapter four

$$\begin{array}{l}\texttt{NEW\_ENV n HEAP[h] ENV[e] e[env]e + n} \\ \quad \Longrightarrow \quad \texttt{ENV[h] h[env]i HEAP[i].}\end{array}$$

is, in canonical form, where variables introduced by desugaring are of the form $\texttt{x}_i$,

$$\begin{array}{l}\texttt{PC[x}_0\texttt{]x}_1\ \texttt{x}_0\texttt{[new\_env]x}_2\ \texttt{x}_2\texttt{[n]x}_3\ \texttt{HEAP[h]x}_4\ \texttt{ENV[e]x}_5 \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \texttt{e[env]e + n} \\ \quad =[\ \texttt{new\_env = NEW\_ENV}\ ]\Rightarrow \\ \qquad \texttt{PC[x}_3\texttt{]x}_6\ \texttt{ENV[h]x}_7\ \texttt{h[env]i HEAP[i]x}_8\texttt{.}\end{array}$$

A similar canonical form can be defined for archetype definitions. The archetype expansion mechanism is so defined that it preserves canonicity.

Implicit Internal Representations. It will not always be explicitly stated that an internal representation is being discussed, this should be clear from the context. For example, the phrase 'left and right locators of the guard' obviously refers to the left and right locators of an internal representation of the guard.

Fixed Offsets. Many of the structures used in the implementation presented here are designed in such a way that the offsets within the structures are fixed, either globally, or for each update scheme. It is then possible to give constants for addressing the constituent parts of a construct, relative to its left locator. For example, access to the values of objects in an update scheme is through the store `Values`. A structure is created in this store for each update scheme (and archetype) containing the relevant values, or pointers to these values. The offset from the left locator of this structure to the left locator of the cell containing (a pointer to) its value is fixed. For an object *obj* let this be the constant $\texttt{O}_{obj}$. If the left locator of the corresponding value structure is `val` then the cells with

left locator $\texttt{val} + \texttt{0}_{obj}$ will contain (a pointer to) *obj*. In all situations, in the implementation presented here, in which such objects are addressed by way of their offset, the left locator of the storage structure will be in a register V. An ambidextrous archetype can be defined to access these values.

$$\texttt{val}(\texttt{0}_{obj}) = \texttt{val} \mid \texttt{V[v]}\ \texttt{v} + \texttt{0}_{obj}\texttt{[val]}.$$

The notation $\texttt{val}^{obj}$ will be used as an abbreviation for $\texttt{val}(\texttt{0}_{obj})$. This convention will also be applied to structures in other stores.

|Locator|. The length of a locator occurs frequently in the update schemes in this chapter. '𝕃' will be used as abbreviation for |Locator|.

Annotation. Where it is not obvious which object is being represented in a value storage structure (for example, because it is represented via an indirection) the text of the corresponding object from the canonical form of the update scheme or archetype definition from which it is taken will be included, in italics, under its representation. A pointer to an object will be indicated by prefixing an indirection symbol ('∗') to the object's text. See example 1(b) for an example.

Standard Environment. This thesis does not specify the standard environment of Update Plans. Where necessary, requirements for the standard environment will be indicated.

Memory Access. As mentioned in chapter five, certain mappings must be defined in the standard environment. One of these is the memory access function. The '@' symbol will be used for the memory access function — i.e. $@(l, n)$ will, when applied to a configuration containing $l[c]l + n$, yield c. The default value of n is 𝕃, so that $@(l, \mathbb{L})$ may be written $@l$. As usual, the @ operator has a higher priority than + or −, so that $@Cell + 3 = (@Cell) + 3$. Again @ can be defined as an archetype.

$$@(\texttt{l},\texttt{n}) = \texttt{val} \mid \texttt{l[val]l} + \texttt{n}.$$

Expression Evaluation. There must be some internal representation of expressions, and some mechanism for evaluating them. The presentation of the implementation must distinguish between the text of expressions, as they appear in update schemes, and the internal representation of expressions. As usual, a typewriter font will be used for the text as it appears in an update scheme. An italic typewriter font will be used to indicate an internal representation of an expression. The evaluation function will be written $\mathcal{E}[\![\cdot]\!]$. E.g. the internal representation of the expression @Cell+3 is $@\mathit{Cell} + \mathit{3}$. Given a configuration satisfying Cell[4] the value of $\mathcal{E}[\![@\mathit{Cell} + \mathit{3}]\!]$ will be @Cell + 3, or 7, the representation of which is *7*. It is assumed that the length of an object can be determined by an examination of its internal representation.

Uninstantiated Values. It will be assumed that there are (possibly typed) special constants representing uninstantiated values. The "don't care" symbol '_' will be used for these constants.

2 Basic Update Schemes

This section defines a straightforward implementation of basic Update Plans. This will be extended in section 3 to cover archetypes. In both this

Stores		Registers	
`Values`	Storage for fixed length values and pointers to values	`V`	Pointer to `Values`
`Heap`	Storage for other values	`H`	Pointer to `Heap`
`Rhs`	Storage for the internal representations of right hand sides	`R`	Pointer to `Rhs`
`Guard`	Storage for internal representation of guards	`G`	Pointer to `Guard`
`NO` Register holding the number of the current scheme			

Figure 1: Stores and registers required for the implementation of basic Update Plans.

section and section 3 the update schemes and archetypes will be considered to be deterministic. In particular, it will be assumed that applicability of archetypes can be decided at the time of their expansion. Nondeterminism and backtracking will be introduced in section 4, and the backtracking mechanism adapted to implement parallel blocks in section 4.3.

Implementing basic Update Plans is simple. Each update scheme is numbered, and a register, `NO`, is added to the machine, in which the number of the update scheme currently under consideration is stored. If the scheme can be applied it is, and the counter in `NO` is reset to 1. If the counter passes the number of schemes in the plan, then no applicable scheme could be found, and execution must stop. This is expressed in the following update schemes, which replace the n^{th} update scheme, $lhs_n \ =[\ guard_n\ ]\Rightarrow\ rhs_n$. N is the number of update schemes in the update plan.

$$\begin{array}{ll} \texttt{NO[no]}\ lhs_n & =[\ guard_n\ ]\Rightarrow\ \texttt{NO[1]}\ rhs_n; \\ \texttt{NO[no]} & =[\ \texttt{n} \leq N\ ]\Rightarrow\ \texttt{NO[no + 1]}. \end{array}$$

(Note the use of alternatives in this scheme.) This in no way changes the semantics of the update plan, it merely makes the sequential consideration of update schemes explicit. A fuller specification of an implementation is given in section 2.2.

2.1 Internal Representation

This section defines, mostly by archetype, the internal representation of update schemes. The stores and registers required by the representations are given in figure 1.

Values. In order to make values accessible they are classified as being either the value of a variable of static length — i.e. one with a length determinable from the update scheme and typing information alone —

or one of a variable of dynamic length. The former are stored in `Values` while the latter are placed in `Heap`. The values in `Heap` are accessed via pointers in `Values`. An archetype can now be defined for each update scheme, which creates a list of the values appearing in the update scheme.

Example 1 (b)

The following archetype defines the structure required for storage of the values in the `NEW_ENV` update scheme, where no is the number assigned to this update scheme.

$$
\begin{array}{l}
\texttt{vals}() = \\
\quad \texttt{N0}[no]\ \texttt{V}[\texttt{val}]\ \texttt{H}[\texttt{hp}] \\
\Longrightarrow \\
\quad \texttt{V}[\texttt{val}']\ \texttt{H}[\texttt{hp}'] \\
\quad \texttt{hp}[\underbrace{@(@\texttt{ENV}, @(@\texttt{PC} + |\texttt{NEW\_ENV}|, |\texttt{Num}|))}_{env}]\texttt{hp}' \\
\quad \texttt{val}[\underbrace{\texttt{PC}}_{PC}\ \underbrace{@\texttt{PC}}_{x_0}\ \underbrace{\texttt{PC} + \mathbb{L}}_{x_1}\ \underbrace{@(@\texttt{PC}, |\texttt{NEW\_ENV}|)}_{new\_env} \\
\qquad \underbrace{@\texttt{PC} + |\texttt{NEW\_ENV}|}_{x_2}\ \underbrace{@(@\texttt{PC} + |\texttt{NEW\_ENV}|, |\texttt{Num}|)}_{n} \\
\qquad \underbrace{@\texttt{PC} + |\texttt{NEW\_ENV}| + |\texttt{Num}|}_{x_3}\ \underbrace{\texttt{HEAP}}_{HEAP}\ \underbrace{@\texttt{HEAP}}_{h} \\
\qquad \underbrace{\texttt{HEAP} + \mathbb{L}}_{x_4}\ \underbrace{\texttt{ENV}}_{ENV}\ \underbrace{@\texttt{ENV}}_{e}\ \underbrace{\texttt{ENV} + \mathbb{L}}_{x_5}\ \underbrace{\texttt{hp}}_{*env} \\
\qquad \underbrace{@\texttt{ENV} + @(@\texttt{PC} + |\texttt{NEW\_ENV}|, |\texttt{Num}|)}_{e+n}\ \underbrace{\texttt{PC} + \mathbb{L}}_{x_6} \\
\qquad \underbrace{\texttt{ENV} + \mathbb{L}}_{x_7}\ \underbrace{@\texttt{HEAP} + @(@\texttt{PC} + |\texttt{NEW\_ENV}|, |\texttt{Num}|)}_{i} \\
\qquad \underbrace{\texttt{HEAP} + \mathbb{L}}_{x_8}]\texttt{val}'.
\end{array}
$$

The expressions for the values of the terms in the update scheme can be derived during grounding analysis.

Note the indirection, via `hp`, to `env`.

A practical implementation would almost certainly instantiate these values one by one, in the order of their dependencies, allowing subexpressions such as `@PC` to be shared. In fact such values would probably not be stored in `Values`, but in a temporary register. The structure presented here should be taken as symbolising internal storage (and access) of values, and not as a definition of a suitable structure for a practical implementation.

Configurations. A locator expression is stored as a triple, containing pointers to its left locator, contents and right locator.

Example 1 (c)

The right hand side of the NEW_ENV update scheme could be defined by the archetype:

$$
\begin{array}{l}
\texttt{rhs}() = \\
\quad \texttt{NO}[no]\ \texttt{R}[\texttt{rhs}]\ \texttt{val}^{\texttt{env}}[\texttt{hp}] \\
\Longrightarrow \\
\quad \texttt{R}[\texttt{rhs}'] \\
\quad \texttt{rhs}[\texttt{val}^{\texttt{PC}} \quad \texttt{val}^{\texttt{x3}} \quad \texttt{val}^{\texttt{x6}} \\
\qquad \texttt{val}^{\texttt{ENV}} \quad \texttt{val}^{\texttt{h}} \quad \texttt{val}^{\texttt{x7}} \\
\qquad \texttt{val}^{\texttt{h}} \quad \texttt{hp} \quad \texttt{val}^{\texttt{i}} \\
\qquad \texttt{val}^{\texttt{HEAP}} \quad \texttt{val}^{\texttt{i}} \quad \texttt{val}^{\texttt{x8}}]\texttt{rhs}'.
\end{array}
$$

Guards. An internal representation for guards must also be available. Precise details will not be given here, since this would entail a full specification of function representation and evaluation in the standard environment. A guard will be assumed to consist of a conjunction of simpler conditions that can be evaluated by some mechanism defined in the standard environment. A guard is then translated to a list of pointers to internal representations of its constituent clauses.

Example 1 (d)

The guard of the NEW_ENV update scheme contains only one clause. The archetype defining the internal representation of this guard is:

$$
\begin{array}{l}
\texttt{grd}(\texttt{grd}, \texttt{grd}', \texttt{val}) = \\
\quad \texttt{NO}[no]\ \texttt{G}[\texttt{grd}]\ \texttt{H}[\texttt{hp}]\ \texttt{val}^{\texttt{new\_env}}[\texttt{new\_env}] \\
\Longrightarrow \\
\quad \texttt{G}[\texttt{grd}']\ \texttt{H}[\texttt{hp}']\ \texttt{grd}[\texttt{hp}]\texttt{grd}' \\
\quad \texttt{hp}[\mathit{new\_env} = \mathit{NEW\_ENV}]\texttt{hp}'.
\end{array}
$$

2.2 An Implementation

The specification is given in terms of phases of an "instruction cycle", in a manner similar to the update schemes in section 5.3 of chapter seven. Each phase may, itself, consist of many steps. The phases are *creation*, *probation*, *verification* and *application*. The archetypes used here are those defined above.

Creation. The creation phase creates the internal representation of the scheme. The constant $\texttt{G}_0$ is the base value of G — i.e. the value G contained in the initial configuration. The creation phase also increments the contents of NO.

[2.1] $\texttt{CREATE NO}[\texttt{no}] \quad =[\ \texttt{n} \leq N\ ]\Rightarrow$
$\texttt{TRY}\ \texttt{G}_0\ \texttt{vals}()\ \texttt{grd}()\ \texttt{rhs}()\ \texttt{NO}[\texttt{no} + 1].$

Probation. Since the update scheme is in canonical form applicability can be checked simply by evaluating the guard. This is done by "walking through" the internal representation of the guard, evaluating its constituent clauses. If all components of the guard have been checked the scheme is applicable. If the current component is true, then the following component must be checked. If it is false then the guard is false and the next scheme must be tried.

[2.2] $\texttt{TRY grd' G[grd']} \quad \Longrightarrow \quad \texttt{VERIFY } R_0\ R_0\texttt{;}$
$\texttt{TRY grd grd[h]grd' h[}\mathit{cond}\texttt{]} \quad =[\ \mathcal{E}[\![\mathit{cond}]\!]\ ]\Rightarrow \quad \texttt{TRY grd';}$
$\texttt{"} \quad \Longrightarrow \quad \texttt{CREATE.}$

Verification. The right hand side must be checked for consistency. This is done by taking each locator expression in turn and checking it for consistency with every other locator expression in the right hand side. The first argument of `VERIFY` is the left locator of the locator expression being checked, the second is that of the locator expression it is being checked against.

The update schemes in this paragraph are a series of alternatives. Commentary has been added to make the intention clearer.
The end of the right hand side has been reached. It is consistent and may be applied.

[2.3] $\texttt{VERIFY rhs' rhs' R[rhs']} \quad \Longrightarrow \quad \texttt{APPLY } R_0\texttt{;}$

The locator expression under consideration has been found to be consistent with the remainder of the right hand side. Move on to the next one.

[2.4] $\texttt{VERIFY } \mathrm{rhs}_1\ \texttt{rhs' R[rhs']}\ \mathrm{rhs}_1[\mathrm{l}_1\ \mathrm{c}_1\ \mathrm{r}_1]\mathrm{rhs}'_1$
$\Longrightarrow \quad \texttt{VERIFY } \mathrm{rhs}'_1\ \mathrm{rhs}'_1\texttt{;}$

The two locator expressions do not overlap and are therefore consistent. Continue verification.

[2.5] $\texttt{VERIFY } \mathrm{rhs}_1\ \mathrm{rhs}_2\ \mathrm{rhs}_1[\mathrm{v}_{\mathrm{l}_1}\ \mathrm{v}_{\mathrm{c}_1}\ \mathrm{v}_{\mathrm{r}_1}]\ \mathrm{rhs}_2[\mathrm{v}_{\mathrm{l}_2}\ \mathrm{v}_{\mathrm{c}_2}\ \mathrm{v}_{\mathrm{r}_2}]\mathrm{rhs}'_2$
$\mathrm{v}_{\mathrm{l}_1}[\mathrm{l}_1]\ \mathrm{v}_{\mathrm{r}_1}[\mathrm{r}_1] \quad \mathrm{v}_{\mathrm{l}_2}[\mathrm{l}_2]\ \mathrm{v}_{\mathrm{r}_2}[\mathrm{r}_2]$
$=[\ \mathrm{r}_1 \le \mathrm{l}_2 \vee \mathrm{r}_2 \le \mathrm{l}_1\ ]\Rightarrow \quad \texttt{VERIFY } \mathrm{rhs}_1\ \mathrm{rhs}'_2\texttt{;}$

They overlap, but the contents of the overlapping cells are consistent.

[2.6] $\texttt{VERIFY } \mathrm{rhs}_1\ \mathrm{rhs}_2\ \mathrm{rhs}_1[\mathrm{v}_{\mathrm{l}_1}\ \mathrm{v}_{\mathrm{c}_1}\ \mathrm{v}_{\mathrm{r}_1}]\ \mathrm{rhs}_2[\mathrm{v}_{\mathrm{l}_2}\ \mathrm{v}_{\mathrm{c}_2}\ \mathrm{v}_{\mathrm{r}_2}]\mathrm{rhs}'_2$
$\mathrm{v}_{\mathrm{l}_1}[\mathrm{l}_1]\ \mathrm{v}_{\mathrm{r}_1}[\mathrm{r}_1] \quad \mathrm{v}_{\mathrm{l}_2}[\mathrm{l}_2]\ \mathrm{v}_{\mathrm{r}_2}[\mathrm{r}_2]$
$\mathrm{v}_{\mathrm{c}_1} + (\max(\mathrm{l}_1, \mathrm{l}_2) - \mathrm{l}_1)[\mathrm{c}]\mathrm{v}_{\mathrm{c}_1} + (\min(\mathrm{r}_1, \mathrm{r}_2) - \mathrm{l}_1)$
$\mathrm{v}_{\mathrm{c}_2} + (\max(\mathrm{l}_1, \mathrm{l}_2) - \mathrm{l}_2)[\mathrm{c}]\mathrm{v}_{\mathrm{c}_2} + (\min(\mathrm{r}_1, \mathrm{r}_2) - \mathrm{l}_2)$
$\Longrightarrow \quad \texttt{VERIFY } \mathrm{rhs}_1\ \mathrm{rhs}'_2\texttt{;}$

The locator expressions are inconsistent. Abort this scheme and try the next one.

[2.7] $\texttt{VERIFY } \mathrm{rhs}_1\ \mathrm{rhs}_2 \quad \Longrightarrow \quad \texttt{CREATE.}$

Application. `APPLY` takes each locator expression in turn from the internal representation and translates it to a concrete configuration. It terminates when it has run through the complete right hand side, and

Stores		Registers	
`Archetypes`	Storage for pointers to as yet unexpanded archetypes	`A`	Pointer to `Archetypes`
`Parameters`	Storage for pointers to archetype parameters	`P`	Pointer to `Parameters`
`Lhs`	Storage for the internal representations of left hand sides	`L`	Pointer to `Lhs`
`Temp`	Temporary storage, in which the left locators of values provisionally instantiated are noted	`T`	Pointer to `Temp`
`F` A frame register, in which the previous contents of the other registers are temporarily stored			

Figure 2: Additional stores and registers for the implementation of archetypes

resets the update scheme number to 1.

[2.8] `APPLY rhs′ R[rhs′]` $\Longrightarrow$ `CREATE NO[1];`

[2.9] `APPLY rhs rhs[`$\mathtt{v_l}$ $\mathtt{v_c}$ $\mathtt{v_r}$`]rhs′` $\mathtt{v_l}$`[l]` $\mathtt{v_c}$`[c]` $\mathtt{v_r}$`[r]`
$\Longrightarrow$ `APPLY rhs′ l[c]r.`

3 Archetypes

Not only can command style update schemes be considered to be the bodies of an archetype definition, as shown in chapter seven, all canonical update schemes can. The "root" archetype then has empty expansions, the whole update scheme appearing in the context. In implementing archetypes all the update schemes in an update plan will be considered to define such an archetype, here called `root`. This archetype has only the implicit parameters $_{[}l$, $l_{]}$, $_{[}r$ and $r_{]}$, and these are irrelevant, since the expansions are empty. An application of update scheme n can now be treated as an application of the n^{th} definition of `root`.

Some additional stores and registers are required for the implementation of archetypes. These are listed in figure 2.

3.1 Structures

Some changes need to be made to the structures defined in section 2.1, and some new structures need to be defined. As the `NEW_ENV` update scheme was the running example in section 2 so will the archetype definition in

example 2(a) be for this section.

Example 2 (a)

The following archetype definition is designed to illustrate the implementation of archetypes, and is not necessarily a definition of a "useful" archetype.

$$\texttt{expr(l + r)} =$$
$$\texttt{a PLUS b a[expr}_1\texttt{(l)] b[expr}_2\texttt{(r)]}$$
$$\Longrightarrow$$
$$\texttt{PLUS expr}_1\texttt{(l) expr}_2\texttt{(r).}$$

The canonical form of this definition is

$$\texttt{expr}({}_{[}l, l_{]}, {}_{[}r, r_{]}, \texttt{l + r}) =$$
$${}_{[}l\texttt{[a]x}_0\ \texttt{x}_0\texttt{[plus]x}_1\ \texttt{x}_1\texttt{[b]}l_{]}$$
$$\texttt{a[expr}_1\texttt{(a, x}_2\texttt{, x}_4\texttt{, x}_5\texttt{, l)]x}_2\ \texttt{b[expr}_2\texttt{(b, x}_3\texttt{, x}_5\texttt{, }r_{]}\texttt{, r)]x}_3$$
$$=[\ \texttt{plus = PLUS}\ ]\Rightarrow$$
$${}_{[}r\texttt{[PLUS]x}_4$$
$$\texttt{x}_4\texttt{[expr}_1\texttt{(a, x}_2\texttt{, x}_4\texttt{, x}_5\texttt{, l)]x}_5\ \texttt{x}_5\texttt{[expr}_2\texttt{(b, x}_3\texttt{, x}_5\texttt{, }r_{]}\texttt{, r)]}r_{]}.$$

Values. If there were no recursive archetypes full expansion of all archetypes would always terminate, and archetypes could therefore be expanded, in a macro-like manner, without reference to the current configuration, and no special mechanism would be required for their implementation. However, archetypes may be recursive, and consequently they must be expanded "on the fly". This means that many values cannot be instantiated immediately, but only after archetypes have been expanded. Another consequence of the archetype mechanism is that values will be shared by way of archetype parameters. This sharing is implemented by adding an extra indirection. All values are now stored on `Heap`, and pointers to these values are maintained in `Values`. These aspects are illustrated in example 2(b), in which as yet uninstantiated values are indicated by the "don't care" symbol '_'.

Example 2 (b)

A value storage archetype for archetype definition 2(a).

$$\texttt{vals()} =$$
$$\texttt{NO[EXPR no] V[val] H[hp]}$$
$$\Longrightarrow$$
$$\texttt{V[val}'\texttt{] H[hp}'\texttt{]}$$
$$\texttt{val}[\underbrace{\_}_{[l}\ \underbrace{\_}_{l_{]}}\ \underbrace{\_}_{[r}\ \underbrace{\_}_{r_{]}}\ \underbrace{\texttt{hp}}_{*l+r}\ \underbrace{\_}_{b}$$
$$\underbrace{\_}_{x_0}\ \underbrace{\_}_{plus}\ \underbrace{\_}_{x_1}\ \underbrace{\_}_{a}\ \underbrace{\_}_{l}\ \underbrace{\_}_{x_2}$$
$$\underbrace{\_}_{r}\ \underbrace{\_}_{x_3}\ \underbrace{\texttt{hp}_1}_{*PLUS}\ \underbrace{\_}_{x_4}\ \underbrace{\_}_{x_5}]\texttt{val}'$$
$$\texttt{hp[@@}val^{\texttt{l}} + \texttt{@@}val^{\texttt{r}}\texttt{]hp}_1\texttt{[}PLUS\texttt{]hp}'.$$

Note that the NO register now also includes an identification of the archetype.

Parameters. A new structure is introduced in which archetype parameters are stored. The first parameter list in the structure is that of the definition. This is followed by parameter lists of archetype calls in the body of the definition. The parameters are addressed through Values.

Example 2 (c)

Parameter storage for archetype definition 2(a). The first five entries in the structure are the parameters of the archetype definition, the next five those of the call $\texttt{expr}_1(\texttt{l})$, and the remainder those of $\texttt{expr}_2(\texttt{r})$.

$$
\begin{array}{l}
\texttt{pars}() = \\
\qquad \texttt{NO[EXPR no] P[par]} \\
\quad \Longrightarrow \\
\qquad \texttt{P[par']} \\
\qquad \texttt{par}[\texttt{val}^{[l} \quad \texttt{val}^{l]} \quad \texttt{val}^{[r} \quad \texttt{val}^{r]} \quad \texttt{val}^{\texttt{l+r}} \\
\qquad\quad \texttt{val}^{\texttt{a}} \quad \texttt{val}^{\texttt{x2}} \quad \texttt{val}^{\texttt{x4}} \quad \texttt{val}^{\texttt{x5}} \quad \texttt{val}^{\texttt{l}} \\
\qquad\quad \texttt{val}^{\texttt{b}} \quad \texttt{val}^{\texttt{x3}} \quad \texttt{val}^{\texttt{x5}} \quad \texttt{val}^{r]} \quad \texttt{val}^{\texttt{r}}\texttt{]par'}.
\end{array}
$$

Left Hand Sides. It now becomes necessary to have an internal representation of the left hand side, since information contained within the left hand side is required to resolve terms. This representation contains two types of structures, one representing ordinary locator expressions, and the other archetype calls. The first field of each type of structure contains a constant identifying its type. The remaining three items of a "LOC" structure are pointers to the left locator, the contents, and the right locator respectively. The second item in an "ARCH" structure identifies the archetype in question. The third points to the call's parameter list, and the fourth will point to the parameter list of the definition when the call is expanded.

Example 2 (d)

$$
\begin{array}{l}
\texttt{lhs}() = \\
\qquad \texttt{NO[EXPR no] L[lhs]} \\
\quad \Longrightarrow \\
\qquad \texttt{L[lhs']} \\
\qquad \texttt{lhs}[\texttt{LOC} \quad \texttt{val}^{[l} \quad \texttt{val}^{\texttt{a}} \quad\quad \texttt{val}^{\texttt{x0}} \\
\qquad\quad \texttt{LOC} \quad \texttt{val}^{\texttt{x0}} \quad \texttt{val}^{\texttt{plus}} \quad \texttt{val}^{\texttt{x1}} \\
\qquad\quad \texttt{LOC} \quad \texttt{val}^{\texttt{x1}} \quad \texttt{val}^{\texttt{b}} \quad\quad \texttt{val}^{l]} \\
\qquad\quad \texttt{ARCH} \quad \texttt{EXPR} \quad \texttt{par}^{\texttt{expr}_1} \quad\quad \_ \\
\qquad\quad \texttt{ARCH} \quad \texttt{EXPR} \quad \texttt{par}^{\texttt{expr}_2} \quad\quad \_\texttt{]lhs'}.
\end{array}
$$

Guards. Guards are implemented as in section 2.

Example 2 (e)

```
grd() =
        NO[EXPR no] G[grd] H[hp] val^plus[plus]
  ⟹
        G[grd'] H[hp'] hp[plus = PLUS]hp' grd[hp]grd'.
```

Right Hand Sides. The representation of right hand sides is similar to that of left hand sides. However there is no need to store archetype calls on the right hand side since these are fully documented in the representation of the left hand side. Consequently it is no longer necessary to indicate the type of the structure (locator expression or archetype call) being stored.

Example 2 (f)

```
rhs() =
        NO[EXPR no] R[rhs]
  ⟹
        R[rhs']
        rhs[val^r val^PLUS val^x4]rhs'.
```

Archetype Calls. The order in which archetypes are expanded depends on the order in which their parameters contribute to grounding — it may be necessary to expand one archetype in order to ground terms necessary for the expansion of another. This order can be determined during grounding analysis.

Example 2 (g)

```
calls() =
        NO[EXPR no] A[a]
  ⟹
        A[a']
        a'[lhs^expr1 lhs^expr2]a.
```

3.2 An Implementation

The following implementation of archetypes assumes that the update plans being implemented are fully deterministic — i.e. that at any point during archetype expansion sufficient information is available (a sufficient set of terms is instantiated) to determine uniquely, by inspection of archetype definitions, which archetype definition, if any, is applicable. Provisional expansion, until the probation phase, is used to make this inspection possible. Values that need to be restored to an uninstantiated value if expansion is unsuccessful are stored in `Temp`.

In keeping with the idea that all applications of update schemes are cases of archetype expansion the creation phase is renamed *expansion*.

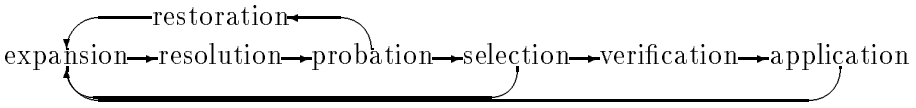


Figure 3: The instruction cycle for the archetype machine.

The initial configuration must satisfy `NO[ROOT 1]`, ensuring that expansion begins with the first update scheme in the update plan. Three new phases are needed in the "instruction cycle", a *resolution phase* in which values are derived for as many terms from the current update scheme as possible, a *restoration phase* in which a provisional expansion is undone if probation is unsuccessful, and a *selection phase*, in which a choice is made between continuing expansion and application. The new instruction cycle is graphically represented in figure 3.

Expansion. The expansion phase constructs an expansion of the current archetype definition and provisionally appends this to the current scheme. The previous values of the pointers to the stores are temporarily stored in the frame register, allowing them to be restored if this expansion is unsuccessful.

[3.1] `EXPAND L[lhs] G[grd] R[rhs] V[val] P[par] A[a] NO[arch no]`
$=[\ \mathtt{n} \leq N^{\mathtt{arch}}\ ]\Rightarrow$
`RESOLVE vals() pars() calls() lhs() grd() rhs()`
`F[lhs grd rhs val par a] NO[arch no + 1].`

Resolution. The resolution phase is not presented in detail, involving as it does details of the expression evaluation mechanism from the standard environment. The resolution phase must:

- by inspection of archetype calls in the representation of the left hand side, match parameters of archetype calls and definitions,
- instantiate any expressions on the heap that become ground,
- examine the left and right hand side to determine if any other terms have become ground,
- note which terms have been instantiated during this resolution phase so that these instantiations can be undone if probation fails.

Example 2 (h)

Given the update scheme

`INF expr(v)` $\Longrightarrow$ `RP expr(v).`

or, in canonical form

$\mathtt{PC[pc]x_0\ pc[inf]x_1\ x_1[expr(x_1, x_2, x_5, x_2, v)]x_2}$
$=[\ \mathtt{inf = INF}\ ]\Rightarrow$
$\mathtt{PC[x_3]x_4\ x_3[RP]x_5\ x_5[expr(x_1, x_2, x_5, x_2, v)]x_2.}$

the terms PC, pc, $\mathtt{x}_0$, inf, $\mathtt{x}_1$, $\mathtt{x}_4$ and RP will be instantiated before any archetype expansion takes place. Given the archetype definition from example 2(a)

$$
\begin{aligned}
&\mathtt{expr}({}_{[}l, l_{]}, {}_{[}r, r_{]}, \mathtt{l} + \mathtt{r}) = \\
&\quad {}_{[}l[\mathtt{a}]\mathtt{x}_0\ \mathtt{x}_0[\mathtt{plus}]\mathtt{x}_1\ \mathtt{x}_1[\mathtt{b}]l_{]} \\
&\quad \mathtt{a}[\mathtt{expr}_1(\mathtt{a}, \mathtt{x}_2, \mathtt{x}_4, \mathtt{x}_5, \mathtt{l})]\mathtt{x}_2\ \mathtt{b}[\mathtt{expr}_2(\mathtt{b}, \mathtt{x}_3, \mathtt{x}_5, r_{]}, \mathtt{r})]\mathtt{x}_3 \\
&= [\ \mathtt{plus} = \mathtt{PLUS}\ ] \Rightarrow \\
&\quad {}_{[}r[\mathtt{PLUS}]\mathtt{x}_4 \\
&\quad \mathtt{x}_4[\mathtt{expr}_1(\mathtt{a}, \mathtt{x}_2, \mathtt{x}_4, \mathtt{x}_5, \mathtt{l})]\mathtt{x}_5\ \mathtt{x}_5[\mathtt{expr}_2(\mathtt{b}, \mathtt{x}_3, \mathtt{x}_5, r_{]}, \mathtt{r})]r_{]}.
\end{aligned}
$$

the first step of archetype expansion — i.e. before any further expansion of archetype calls in the archetype body takes place — will in turn instantiate ${}_{[}l$, a, $\mathtt{x}_0$, $\mathtt{x}_1$, plus, $l_{]}$, b and $r_{]}$ from the archetype's body, and $\mathtt{x}_2$ from the update scheme in which it is called. Assuming that the left locator of the value storage structure of the update scheme is $\mathtt{v}_0$, and that of the storage structure of the archetype $\mathtt{v}_1$, immediately after the resolution phase of this cycle Temp will satisfy the configuration (with a slight abuse of notation)

$$\mathtt{T}[\mathtt{t}]\ \mathtt{t}[\mathtt{v}_0^{\mathtt{x}_2}\ \mathtt{v}_1^{[l}\ \mathtt{v}_1^{\mathtt{a}}\ \mathtt{v}_1^{\mathtt{x}_0}\ \mathtt{v}_1^{\mathtt{x}_1}\ \mathtt{v}_1^{\mathtt{plus}}\ \mathtt{v}_1^{l_{]}}\ \mathtt{v}_1^{\mathtt{b}}\ \mathtt{v}_1^{r_{]}}]\mathtt{T}_0$$

where $\mathtt{T}_0$ is the base value of T — i.e. the value in T in the initial configuration.

Probation. It is now possible, even likely, that terms in a guard are not instantiated. The evaluation mechanism must be able to handle uninstantiated values, returning a "don't care" value for any expression containing an uninstantiated value. Probation only fails if a condition is false. If the guard is non-false the current expansion is successful, and expansion may proceed. If probation fails the store pointers must be reset to their old values, and the restoration phase entered in order to undo any instantiations.

[3.2] TRY grd′ G[grd′] $\Longrightarrow$ SELECT;
TRY grd grd[h]grd′ h[*cond*] $= [\ \mathcal{E}[\![\mathit{cond} \neq \mathit{FALSE}]\!]\ ] \Rightarrow$ TRY grd′;
TRY F[lhs grd rhs val par a]
$\Longrightarrow$ RESTORE L[lhs] G[grd] R[rhs] V[val] P[par] A[a].

Restoration. The restoration phase must undo any instantiations from the preceding resolution phase.

[3.3] RESTORE T[$\mathtt{T}_0$] $\Longrightarrow$ EXPAND;
RESTORE T[t] t[v]t′ $\Longrightarrow$ RESTORE T[t′]′ v[_].

Selection. The selection phase inspects the archetype stack to determine if archetype expansion is complete. If A points to the bottom of the archetype stack, here indicated by the constant A_0, then expansion is complete, and verification may commence. If not, the following archetype must be popped from the archetype stack, and the parameter list of the archetype's definition (which will be created by the expansion phase) connected to the archetype call on the left hand side.

[3.4] SELECT A[A_0] $\Longrightarrow$ VERIFY R_0 R_0;
SELECT A[a] a[l]a′ l[ARCH arch call _] P[def]
$\Longrightarrow$ EXPAND A[a′] NO[arch 1] l[ARCH arch call def].

Verification. Some indirections have been added to the internal representation of the right hand side. If these are taken into account the verification phase is identical to that in section 2. For example, update scheme 2.5 becomes

$$\begin{array}{l}\texttt{VERIFY rhs}_1\ \texttt{rhs}_2\ \texttt{rhs}_1[\texttt{v}_{\texttt{l}_1}\ \texttt{v}_{\texttt{c}_1}\ \texttt{v}_{\texttt{r}_1}]\ \texttt{rhs}_2[\texttt{v}_{\texttt{l}_2}\ \texttt{v}_{\texttt{c}_2}\ \texttt{v}_{\texttt{r}_2}]\texttt{rhs}_2' \\ \qquad\qquad \texttt{v}_{\texttt{l}_1}[\texttt{h}_{\texttt{l}_1}]\ \texttt{v}_{\texttt{r}_1}[\texttt{h}_{\texttt{r}_1}]\ \ \texttt{v}_{\texttt{l}_2}[\texttt{h}_{\texttt{l}_2}]\ \texttt{v}_{\texttt{r}_2}[\texttt{h}_{\texttt{r}_2}] \\ \qquad\qquad \texttt{h}_{\texttt{l}_1}[\texttt{l}_1]\ \texttt{h}_{\texttt{r}_1}[\texttt{r}_1]\quad\ \texttt{h}_{\texttt{l}_2}[\texttt{l}_2]\ \texttt{h}_{\texttt{r}_2}[\texttt{r}_2] \\ =[\ \texttt{r}_1 \le \texttt{l}_2 \vee \texttt{r}_2 \le \texttt{l}_1\ ]\Rightarrow\ \ \texttt{VERIFY rhs}_1\ \texttt{rhs}_2'; \end{array}$$

Application. Application is again, when adapted to the extra indirections, more or less identical to the application phase of section 2. Upon completion, application must now call EXPAND NO[ROOT 1] rather than CREATE NO[1].

APPLY rhs′ R[rhs′] $\Longrightarrow$ EXPAND NO[ROOT 1];

4 Backtracking

Nondeterminism in update plans can have three sources. Firstly, more than one update scheme in the plan may be applicable. This is the simplest form to construct a backtracking mechanism for. Secondly, an update scheme containing semi-ground terms may have more than one applicable instantiation. Finally, more than one archetype expansion may be possible. An approach to backtracking across ambiguities of the first type is presented, in terms of the specification of basic Update Plans, in section 4.1. Adapting this to update plans with deterministic archetypes and no semi-ground terms would be simple. Section 4.2 discusses adapting the backtrack mechanism to cover semi-ground terms and nondeterministic archetypes. Finally section 4.3 shows how the backtrack mechanism can be used to implement parallel blocks.

4.1 Basic Backtracking

The symmetry of update schemes suggests that backtracking can be achieved by simply reversing update schemes — by swapping the left and right hand sides. Indeed this is almost the case. Reversing update schemes is the kernel of the backtracking mechanism. However, including the guard in the reversal is unnecessary, and in fact erroneous. Erroneous since a guard that was true before application is not necessarily true after application, and therefore before backtracking. Unnecessary because

backtracking is applied deterministically. The update scheme used for backtracking across update scheme lhs =[$guard$]⇛ rhs is rhs⟹lhs. This is called the *converse* of lhs =[$guard$]⇛ rhs.

Example 3 (a)

The converse of the update scheme

[4.1] PC[pc] pc[MOV r_1 r_2]qc r_1[x]⟹PC[qc] r_2[x].

is

[4.2] PC[qc] r_2[x]⟹PC[pc] pc[MOV r_1 r_2]qc r_1[x].

The conceptual structure of backtracking is as follows. The numbering from section 2 is applied, using the register NO, and a backtrack stack, addressed by the register B, is introduced. Each update scheme is then replaced as follows. If the update scheme with index n is lhs_n =[$guard_n$]⇛ rhs_n then it is implemented in the backtracking version of the update plan as:

$$\begin{array}{lll} \texttt{NO}[n]\ lhs_n^*\ \texttt{B}[\texttt{c}] & =[\ guard_n\ ]\Rrightarrow & \texttt{NO}[1]\ rhs_n\ \texttt{B}[\texttt{b}]\ \texttt{b}[n\ \alpha_n]\texttt{c}; \\ \texttt{NO}[n] & \Longrightarrow & \texttt{NO}[\texttt{n}+1 \bmod \texttt{N}+1]. \end{array}$$

$$\texttt{NO}[0]\ rhs_n\ \texttt{B}[\texttt{b}]\ \texttt{b}[n\ \alpha_n]\texttt{c} \Longrightarrow \texttt{NO}[\texttt{n}+1 \bmod \texttt{N}+1]\ lhs_n^*\ \texttt{B}[\texttt{c}].$$

where N is the number of update schemes in the update plan. The first scheme applies scheme n, if possible, and pushes any information necessary for backtracking onto the backtrack stack, and starts looking for schemes applicable to the new configuration by resetting NO to one. Its alternative simply moves on to the next possibility. The last scheme backtracks across an application of scheme n. The significance of the superscript $*$, and the definition of α_n, are given below. Informally the superscript indicates a simple transformation of the original update scheme, necessary to ensure correct backtracking, and α_n represents information that cannot be derived from the converse of the original update scheme, and that, therefore, must be preserved on the backtrack stack.

Completed Left Hand Sides. The superscript $*$ is used to indicate *completed left hand sides*. To ensure proper backtracking the left hand side of an update scheme must be extended to cover cells that are updated by the original update scheme but not specifically covered by the left hand side. This extension, in combination with pushing unprotected variables (see the next section), ensures that the contents of all cells will be properly restored. The left hand side is said to be *completed* with respect to the right hand side.

Example 3 (b)

The update — i.e. the unconditional update rule — specified by 4.1 is

[4.3] ⟹PC[qc] r_2[x].

(The variables appearing in 4.1 are here used to represent their values in the current configuration.) The update specified by 4.1's converse, 4.2, is

[4.4] $\Longrightarrow$PC[pc] pc[MOV r_1 r_2]qc r_1[x].

Combining these, i.e. applying first 4.1 and then 4.2, gives

$\Longrightarrow$PC[pc] pc[MOV r_1 r_2]qc r_1[x] r_2[x].,

which contains the locator expression r_2[x], which is not on the left hand side of 4.1 — i.e. the net effect of 4.3 and 4.4 on a configuration in which 4.1 is applicable is to update the contents of r_2 to x.

Fortunately locator expressions such as r_2[x] in example 4(b) are easy to detect. Two locator expressions are said to *correspond* if they have the same left and right locators. Any locator expression on the right hand side of an update scheme without a corresponding locator expression on the left hand side will not be restored by backtracking using only converses. For each locator expression $l[x]r$ on the right hand side of an update scheme for which there is no corresponding expression on the left hand side, the locator expression $l[y]r$ must be added to the left hand side, where y is a new variable.

Example 3 (c)

Applying this extension to update scheme 4.1 gives

[4.5] PC[pc] pc[MOV r_1 r_2]qc r_1[x] r_2[y]
$\Longrightarrow$PC[qc] r_2[x].

A left hand side, *lhs*, completed in this way is written *lhs**. The fact that y is non-ground in the converse of 4.5 will be addressed in the next section.

Unprotected Variables. Backtracking by application of a converse is only sure to work correctly if all terms appearing on the right hand side of the converse are fully ground. There is no guarantee that a semi-ground term will be restored to its original value. Indeed, as shown above, the converse may contain semi-ground terms or non-ground terms. Any semi-ground variable in a converse is said to be *unprotected*. Unprotected variables can easily be detected by the grounding mechanism.

Example 3 (d)

The converse of the extended scheme above is

PC[qc] r_2[x]
$\Longrightarrow$PC[pc] pc[MOV r_1 r_2]qc r_1[y].

in which the variable y on the right hand side is unground.

The simplest solution to this problem is to push any unprotected variables onto the backtrack stack. For each update scheme the sequence α_n is defined to be some fixed sequence containing exactly the unprotected variables of that update scheme.

The extended converse of an update scheme, combined with preservation of unprotected variables on the backtrack stack is sufficient to ensure correct backtracking. Backtracking can however be considerably simplified by application of the *backtrack rule*.

The Backtrack Rule. Completing left hand sides as described above is often unnecessary, causing the backtrack mechanism to preserve much superfluous information on the backtrack stack such as, for example, the value contained in the cell just above a stack when the stack is expanded by a push operation.

Example 4

The backtrack version of the update scheme

```
PUSH x SP[t]  ⟹  s[x]t SP[s].
```

is

```
PUSH x NO[no] SP[t] s[y]t B[c]
  ⟹  NO[1] s[x]t SP[s] B[b] b[n y]c.
```

in which the value of `y` is, assuming that `SP` really does address a stack, unnecessarily pushed onto the backtrack stack.

A notational convention is needed to indicate changes that need not be reversed on backtracking. It is, of course, the responsibility of the update plan writer to ensure that the semantics of the plan will not be affected by use of this convention, though memory structure analysis as described in chapter five may be of some help. Meijer [53] introduced such a notational convention, known as the *backtrack rule*. The backtrack rule states

> *all cells that need to be restored upon backtracking are explicitly mentioned (covered) in the left hand side.*

It is the task of the update plan writer to ensure that an update plan satisfies the backtrack rule. Application of the backtrack rule is equivalent to omitting the extension of the left hand side. In the specification of backtracking above any lhs^* may be replaced by lhs, to give a specification of backtracking with the backtrack rule applied.

4.2 Extended Backtracking

There is not a lot to be said about extending backtracking to cover semi-ground terms and ambiguous archetypes. In the first case since the ambiguity is at the level of the expression evaluation mechanism, and in the second since the implementation described here takes a brute force approach. In both cases the backtrack mechanism in section 4, adapted to the implementation in section 3 will be the departure point. In particular it will be assumed that a representation of both the left and right hand side of the update scheme being "undone" will have been constructed.

Semi-Ground Terms. The expression evaluation mechanism from the standard environment must be able to determine from examination of the values of semi-ground terms in the representation of the update scheme if all possible sets of values of have been tried. If all possible instantiations have been exhausted execution continues with the next update scheme. If not, the current update scheme (and archetype expansion) must be tried with the next possible instantiation.

If the values alone do not provide sufficient information for the backtracking mechanism of expression evaluation, sufficient data must be pushed to the backtrack stack to enable evaluation to recommence where it left off.

Ambiguous Archetypes. Unfortunately it is not clear how to implement efficient backtracking across ambiguous archetypes. A brute force approach is to reverse the complete expansion of the update scheme, and to determine the next applicable expansion, if any, from inspection of the expansion history which will at that point still be available immediately above the top of the backtrack stack, and to re-expand. An alternative approach is to treat each archetype definition as an independent update scheme, and to determine a minimal set of terms that guarantees grounding of the converse. This set of terms must then be pushed to the backtrack stack. It is a question for further investigation which of these approaches is the least inefficient.

4.3 Parallel Blocks

Parallel blocks can be implemented by "switching off" the application phase within a parallel block, and applying the techniques from the backtrack mechanism, within the parallel block, to construct a monolithic right hand side consisting of all the right hand sides of all applicable instantiations of update schemes within the parallel block. The same technique can be used on backtracking.

5 Conclusions

A possible implementation of update plans has been discussed. More efficient implementations are certainly possible. In particular, the mechanism for backtracking across archetypes is unsatisfactory in that a new expansion will repeat much of the work that has been undone in the backtracking stage. The efficiency of parallel blocks could also probably be greatly improved by sharing common substructures in the right hand side, again especially if the parallel block contains ambiguous archetypes. The greatest gain in efficiency, however, is in practice probably to be found in analysis of update schemes and archetypes to determine which terms contribute to the applicability of the schemes, and instantiating and checking these terms as soon as possible. In command style schemes, for example, it would be advisable to instantiate and check the value of the contents of the cell addressed by the programme counter before instantiating the rest of the scheme. The same technique can be applied to command archetypes, except that in this case it is the contents of the cell with left locator $\lfloor l$ that must be checked. This requires $\lfloor l$ to be

instantiated at the time, but in most applications of command archetypes this will be the case.

Chapter Eleven Conclusions and Suggestions for Further Research

In this thesis the syntax and semantics of Update Plans have been described. Some extensions to Update Plans (then called 'Update Schemes') as defined by Meijer [53] have been introduced. The thesis also presented some applications of Update Plans, both in specifying abstract and concrete machines, and in proving semantic properties of (programmes for) these machines. This chapter very briefly summarises the findings of this thesis, and presents some possible topics for future research.

1 Conclusions

Update Plans provide a readable and flexible specification formalism for low level languages. Their declarative style makes reasoning about such specifications relatively simple. The addition of the archetype mechanism greatly increases the expressive power of Update Plans, making a form of structured programming possible. The archetype mechanism also introduces new possibilities, such as simple specification of asynchronous parallel processors. Synchronous parallelism can also be specified by making use of the parallel block mechanism.

2 Further Research

Future research should take place on four fronts. Firstly, Update Plans should be placed in a wider theoretical context. Secondly, non-trivial applications should be undertaken, with an eye to developing pragmatics for update plan specifications. Thirdly, on a related note, Update Plans show promise as a didactic tool. This aspect should be further investigated. Finally a full implementation of Update Plans should be developed.

2.1 Theoretical Context

Rewrite Systems. Update Plans form an abstract rewrite system. Work on their relation to other ARS's, and to graph rewrite systems in particular should lead to insights which would allow the well formedness conditions to be relaxed while still guaranteeing one or more of, for

example

- finite ambiguity
- the strong Church-Rosser property
- the weak Church-Rosser property
- modularity

Categories. Update Plans also define a category having consistent configurations as its objects, and update rules as its morphisms. Further investigation may make a category theoretical description of Update Plans possible.

2.2 Applications

Concrete Machines. The Update Plan specification formalism needs to be validated on a set of major sample applications, in order to develop the pragmatics of use. Among these applications are those in which *parallelism* plays a dominant rôle, e.g. the full specification of a "real" processor, such as SUN's Sparc processor, Digital's Alpha [2] or IBM's PowerPC, or the (abstract) implementation of network protocols based on various communication primitives. Update Plans may also be applied to the specification of newer paradigms such as transport-triggered architectures [14, 30, 31, 32].

Abstract Machines. The archetype mechanism enables abstraction away from detail. An interesting challenge is to develop a suitable set of macros, possibly in combination with some simple transformations, such that the surface level language defined is a high level declarative formalism, for example the λ-calculus or some combinatory logic, while the fully expanded update schemes define a concrete implementation.

Hardware Specification. Update Plans could also be applied to the specification of hardware. Some preliminary work has been done in specifying elementary VLSI components [51].

Metrics. An annotation can be developed to indicate the cost of applying an update scheme or archetype. This would make it possible to apply Update Plans to the problem of compiler optimisation. The existence of a working implementation would make this even more useful, since the cost of proposed solutions could then be calculated by means of a simulation.

Formal Verification. For a specification and programming language to be useful at all it is imperative that one may prove a programme correct (e.g. equivalent to a specification) and/or derive a programme from a specification and/or prove that a specification or programme has certain desirable properties. Therefore a verification, and possibly transformation method should be developed, together with appropriate heuristic rules. Chapter six is only a first step in this direction.

2.3 Didactic Applications

The wide applicability and intuitive semantics of Update Plans make them well suited as a didactic tool for courses on machine architectures.

Update Plans should be described and explained in tutorials, containing many examples, in order to "market" and popularise them. A longer term project would be a survey of machine architectures presented in terms of update plans. Update Plans have been successfully used in the compiler construction course given at the University of Nijmegen [67], and in the machine architecture course at the University of Utrecht.

2.4 Implementation

A full implementation should be developed, embedded in an integrated collection of software, comprising at least a development environment, a compiler and a debugger.

The formalism should be further extended, in particular by introducing modules of some sort, possibly based on the archetype mechanism, in order to facilitate abstraction for information hiding, structure reuse, clarification, etc. Also, the parallel blocks introduced in chapter nine do not go much further than replacing nondeterministic execution of an applicable update scheme with simultaneous execution of all applicable update schemes. Such an approach will in practice, however, be too simple-minded. A host of artificial semaphore-like constructs would need to be introduced in order to maintain well-behaviour. Therefore it seems imperative to classify the different ways in which the concept of parallelism is used (e.g. co-operating processes, or systolic processes, or synchronous processes).

Summary

A major concern of computer science is the development of methods for proving software correct — either by verifying programmes already written or, preferably, by designing methodologies that ensure that only correct programmes are written. Much progress has been made, and since compilers themselves are programmes, application of the insights gained should make it possible to guarantee the correctness of compilers. Unfortunately compilers usually produce low level code, e.g. assembler, and there is seldom a formal definition of the semantics of such low level languages. Formal proofs cannot be based on informal descriptions. Update Plans are intended to fill this need. Update Plans constitute a formalism for the specification of low level languages. This is the *low level* part of the title of this thesis. The *high level* part refers to the formalism itself. Update Plans exhibit many of the features of other modern high level programming languages, giving them high abstractive power. Update Plans are a high level language for the specification of low level activities.

This thesis presents a complete formal definition of the syntax and semantics of Update Plans, and several examples of the formalism's use. It also presents a new typing system for Update Plans, and discusses the application of this to the detection of certain common types of memory use. These memory use paradigms have certain semantic properties, and this is also investigated. The thesis also introduces some extensions to Update Plans, in particular a macro-like mechanism, *archetypes*, and a degree of parallelism.

The archetype mechanism increases the expressive power of Update Plans, by significantly increasing their abstractive power, making it easier to write intuitive specifications. The archetype mechanism is illustrated by a specification of intermediate code such as may be emitted by a compiler before register allocation takes place. This is an interesting case in that the language specified is infinite, commands having a tree like structure in which subtrees represent the computation of intermediate results.

The addition of parallelism again extends the expressivity of Update

Plans to cover such features of low level architectures as pipelining. This is illustrated in a partial specification of the Berkeley RISC machine.

The formalism's usefulness, and popularity, will be increased by its realisation as a full programming language. Some aspects of such an implementation are also covered.

Further research into Update Plans could take place on the following fronts.

Update Plans must be developed into a full programming paradigm by the addition of some type of module mechanism.

A complete implementation of Update Plans, embedded in a software environment comprising supporting utilities, a (transformational) development tool, an interpreter and a debugger, should be developed.

The formalism should be further extended to cover, for example, different forms of parallelism.

The relation between Update Plans and (graph) rewrite systems should be investigated — in particular to what extent results from the latter "carry over" to the former.

Finally "real world" problems should be attacked. Full specifications of concrete machine architectures must be produced to illustrate the full power of the formalism. Related to this is work on Update Plans as a didactic tool. Course material should be written (informally) introducing Update Plans, and using them to introduce students to (various types of) machine architecture.

Samenvatting

Een van de hoofddoelen van de informatica is het ontwikkelen van methoden om de correctheid van *software* te bewijzen — óf door verificatie van al geschreven programma's óf, nog beter, door het ontwerpen van methodieken die ertoe leiden dat alleen correcte programma's geschreven worden. Veel vooruitgang is al geboekt, en omdat *compilers* ook zelf programma's zijn kunnen deze methoden ook daarop toegepast worden. Dit zou het mogelijk moeten maken om ook de correctheid van *compilers* te bewijzen, maar helaas is het eindprodukt van een *compiler* meestal laag-nivo code, bijv. *assembler*, waarvan meestal geen formele specificatie voorhanden is. Formele bewijzen kunnen niet steunen op informele beschrijvingen. Update Plans zijn bedoeld om deze leemte te vullen. Update Plans zijn een formalisme voor het specificeren van laag-nivo programmeertalen. Dit is het *low level* deel van de titel van dit proefschrift. Het *high level* deel slaat op het formalisme zelf. Als programmeertaal hebben Update Plans veel gemeen met andere moderne hoog-nivo programmeertalen. Update Plans zijn een hoog-nivo taal voor het specificeren van laag-nivo activiteiten.

Dit proefschrift bevat een volledig formele definitie van de syntaxis en semantiek van Update Plans, geïllustreerd door enkele voorbeelden van hun toepassing. Een nieuw typeringssysteem wordt geïntroduceerd, en de toepassing daarvan m.b.t. het opsporen van bepaalde vaak voorkomende vormen van geheugengebruik wordt beschreven. Deze geheugengebruik paradigma's hebben bepaalde semantische eigenschappen, en dit wordt ook onderzocht. Ook beschrijft het proefschrift uitbreidingen van Update Plans, met name *archetypes*, een *macro*-achtig mechanisme, en een zekere mate van parallellisme.

Het *archetype* mechanisme verhoogt de uitdrukkingskracht van Update Plans door hun abstraherend vermogen significant te verhogen, waardoor het makkelijker wordt intuïtief specificaties te schrijven. Het mechanisme wordt geïllustreerd met een specificatie van tussencode zoals door een *compiler* geproduceerd zou kunnen worden, voordat *register allocation*

plaats vindt. Dit voorbeeld is interessant omdat de gespecificeerde taal oneindig is. Commando's hebben een boomachtige structuur waarin onderbomen het berekenen van tussenwaardes voorstellen.

Het toevoegen van parallellisme verhoogt wederom de uitdrukkingskracht van Update Plans, die daardoor ook zulke eigenschappen van laag-nivo architecturen als *pipelining* kunnen beschrijven. Dit wordt geïllustreerd door een partiële specificatie van de Berkeley RISC machine.

De bruikbaarheid, en populariteit, van Update Plans zal verhoogd worden wanneer ze als volledige programmeertaal gerealiseerd worden. Enkele aspecten van zo'n implementatie worden ook behandeld.

Toekomstige onderzoek op het gebied van Update Plans zou als volgt plaats kunnen vinden.

Update Plans moeten verder ontwikkeld worden, tot een volledig programmeerparadigma, door het toevoegen van een module mechanisme.

Er moet een volledige implementatie van Update Plans komen, met ondersteunende faciliteiten, (transformationeel) ontwikkelingsgereedschap, een vertaler en een *debugger*.

Het formalisme moet verder uitgebreid worden, bijvoorbeeld om verschillende vormen van parallellisme te beschrijven.

Het verband tussen Update Plans en (graph-) herschrijfsystemen moet onderzocht worden — in het bijzonder de mate waarin resultaten in het tweede "geërfd" worden door het eerste.

Ten slotte moeten problemen uit de "echte wereld" aangepakt worden. Volledige specificaties van concrete machine architecturen zouden geschreven moeten worden om de volle kracht van Update Plans te illustreren. Hieraan verwant is aandacht voor Update Plans als didactisch middel. Collegedictaten zouden geschreven moeten worden waarin Update Plans (informeel) uitgelegd worden om vervolgens gebruikt te worden om studenten kennis te laten maken met machine-architecturen.

Curriculum Vitæ

19th January 1956 Date of birth. Ashbourne, UK.

4th July 1977 BSc Mathematics (Div. II), University of Manchester

28th April 1989 "Doctoraal" Computer Science (*cum laude*), University of Nijmegen

1st May 1989 – 30th September 1991 Research Assistant, Esprit Basic Research Action no. 3147, the *Phœnix* project, University of Nijmegen

1st October 1991 – 16th December 1993 "Assistent in Opleiding" (Research Assistant), University of Nijmegen

1st March 1994 – 31st July 1994 Systems Developer, University of Nijmegen

1st October 1994 – 28th February 1995 Consultant, University of Nijmegen

from 1st March 1995 Research Assistant, University of York

Other important dates:

- 18th October 1989
- 11th September 1991

Appendix I Bibliography

The numbers between brackets are the reference numbers of the works cited, the numbers in parenthesis the numbers of the pages on which they are cited.

[1] (4) Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman.
Compilers, Principles, Techniques and Tools.
Addison-Wesley, 1986.

[2] (116) Communications of the ACM, 26(2), February 1993.
(Special issue on the Alpha chip).

[3] (41) K.R. Apt and G.D. Plotkin.
A Cook's tour of countable nondeterminism.
In *Proceedings of the 8th. Colloquium on Automata, Languages and Programming*, volume 115 of *Lecture Notes in Computer Science*. Springer Verlag, 1981.
S. Even and O. Kariv (editors).

[4] (41) R.J.R. Back.
A continuous semantics for unbounded nondeterminism.
Theoretical Computer Science, 23:187–210, 1983.

[5] (1) R.C. Backhouse, P.J. de Bruin, G. Malcolm, E. Voermans, and J. van der Woude.
Relational catamorphisms.
In B. Möller, editor, *Proceedings of the IFIP*

TC2 Working Conference on Constructing Programs from Specifications, pages 287–318, Amsterdam, 1991. North-Holland Publishing Company.

[6] (2) H.P. Barendregt, M.C.J.D. van Eekelen, J.R.W. Glauert, J.R. Kennaway, M.J. Plasmeijer, and M.R. Sleep.
Term graph rewriting.
In *Proceedings of Parallel Architectures and Languages in Europe (PARLE), Volume II: Parallel Languages*, volume 259 of *Lecture Notes in Computer Science*. Springer Verlag, Eindhoven, The Netherlands, 1987.
J.W. de Bakker, A.J. Nijman and P.C. Treleaven (editors).

[7] (1) F.L. Bauer, R. Berghammer, M. Broy, W. Dosch, F. Geiselbrechtinger, R. Gnatz, E. Hangel, W. Hesse, B. Krieg-Brückner, A. Laut, T. Matzner, B. Möller, F. Nickl, H. Partsch, P. Pepper, K. Samelson, M. Wirsing, and H. Wössner.
The Munich Project CIP. Volume I: The Wide Spectrum Language CIP-L, volume 183 of *Lecture Notes in Computer Science*.
Springer-Verlag, Berlin/Heidelberg/New York, 1985.

[8] (1) F.L. Bauer and H. Wössner.
Algorithmic Language and Program Development.
Springer-Verlag, Berlin, 1982.

[9] (1) R.M. Burstall and J. Darlington.
A transformation system for developing recursive programs.
Journal of the ACM, 24(1):44–67, January 1977.

[10] (2) Volker Claus, Hartmut Ehrig, and Grzegorz Rozenburg, editors.
Graph Grammars and their Aplications to Computer Science and Biology, volume 73 of *Lecture Notes in Computer Science*.
Springer Verlag, 1979.

[11] (38) Implementation of CLEAN on MAC II-fx, see [72].

[12] (5) Todd A. Cook, Paul D. Franzon, Ed Harcourt, and Thomas K. Miller III.
System-level specification of instruction sets.
In *Proceedings of the 1993 IEEE International Conference on Computer Design*, pages 552–557, 1993.

[13] (5) Todd A. Cook and Ed Harcourt.
A functional specification language for instruction set architectures.
In *Proceedings of the IEEE International Conference on Computer Languages*, pages 11–19, 1994.

[14] (116) Henk Corporaal and Hans (J.M.) Mulder.
MOVE: A framework for high-performance processor design.
In *Supercomputing-91*, Albuqerque, New Mexico, USA, November 1991.

[15] (5) J.W. Davidson and C.W. Fraser.
The design and application of a retargetable peephole optimizer.
ACM Transactions on Programming Languages and Systems, 2(2):191–202, April 1980.

[16] (5) J.W. Davidson and C.W. Fraser.
Eliminating redundant object code.
In *Conference Record of the 9th Annual ACM Symposium on Principles of Programming Languages*, Albuqerque, New Mexico, USA, 1982.

[17] (30) N. G. de Bruijn.
Lambda calculus notations with nameless dummies, a tool for automatic formula manipulation.
Indagationes Mathematicæ, proceedings of the Koninklijke Nederlandse Akademie der Wetenschappen, pages 381–392, 1972.

[18] (55) Pierre Deransart, Martin Jourdan, and Bernard Lorho.
Attribute grammars : definitions, systems, and bibliography, volume 323 of *Lecture Notes in Computer Science*.
Springer-Verlag, Berlin, 1988.

[19] (4) Digital Equipment Corporation. *PDP-11 Processor Handbook*, 1971.

[20] (1) Charles Donnelly and Richard Stallman. *BISON: The YACC-compatible Parser Generator*, 1988.

[21] (2) Hartmut Ehrig, Manfred Nagl, and Grzegorz Rozenburg, editors. *Graph-Grammars and their Applications to Computer Science*, volume 153 of *Lecture Notes in Computer Science*. Springer Verlag, 1983.

[22] (4) Jon Fairbairn and Stuart Wray. Tim: A simple, lazy abstract machine to execute supercombinators. In *Functional Programming Languages and Computer Architecture*, volume 274 of *Lecture Notes in Computer Science*. Springer Verlag, 1987. Gilles Kahn (editor).

[23] (55) Gilberto Filè. *Theory of Attribute Grammars*. PhD thesis, Technische Hogeschool Twente, 1983.

[24] (69) H. Franzen and B. Hoffman. Automatic determination of affix flow in extended affix grammars. In K. H. Böhling and P. P. Spies, editors, *Informatik-Fachberichte*, Berlin, BRD, 1979. Gesellschaft für Informatik (GI), Springer Verlag.

[25] (1) H. Franzen, B. Hoffman, and I.-R. Petersen. Ein Parser-Generator für erweiterte Affix-Grammatiken. Master's thesis, Technische Universität Berlin, Fachbereich Informatik, October 1976. Bericht Nr. 76-24.

[26] (5) C.W. Fraser. A compact, machine-independent peephole optimizer. In *Conference Record of the 6th Annual*

ACM Symposium on Principles of Programming Languages, San Antonio, Texas, USA, 1979.

[27] (5) Robert Giegerich.
A formal framework for the derivation of machine-specific optimizers.
ACM Transactions on Programming Languages and Systems, 5(3):478–498, July 1983.

[28] (71) Robert Giegerich.
Implementierung von programmiersprachen.
Technical report, Technische Fakultät, Universität Bielefeld, Postfach 86 40, 4800 Bielefeld 1, BRD, 1992.
Lecture notes for a course in compiler construction. (In German).

[29] (5) Ed Harcourt, Jon Mauney, and Todd Cook.
Functional specification and simulation of instruction set architectures.
In *Proceedings of the International Conference on Simulation and Hardware Description Languages*, pages 3–8. The Society for Computer Simulation, 1994.

[30] (116) Jan Hoogerbrugge.
Software pipelining for transport-triggered architectures.
Master's thesis, Delft University of Technology, Delft, The Netherlands, December 1991.

[31] (116) Jan Hoogerbrugge and Henk Corporaal.
Comparing software pipelining for an operation-triggered and a transport-triggered architecture.
In *Compiler Construction: 4th International Conference, CC '92*, volume 641 of *Lecture Notes in Computer Science*. Springer Verlag, Paderborn, BRD, October 1992.
U. Kastens and P. Pfahler (editors).

[32] (116) Jan Hoogerbrugge, Henk Corporaal, and Hans Mulder.
Software pipelining for transport triggered

architectures.
In *Proceedings of the 24th Annual Workshop on Microprogramming*, pages 74–81, Albuqerque, New Mexico, USA, November 1991.

[33] (1) S.C. Johnson.
YACC - Yet Another Compiler-Compiler.
Computer Science Technical Report No. 32, Bell Laboratories, Murray Hill, New Jersey, 1975.

[34] (1) U. Kastens, B. Hutt, and E. Zimmerman.
GAG: A Practical Compiler Generator.
Springer-Verlag, 1982.

[35] (90) Manolis G.H. Katevenis.
Reduced Instruction Set Architectures for VLSI.
ACM Doctoral Dissertation Awards. The MIT Press, 1985.

[36] (17,II.1) P. Klint.
The ASF+SDF Meta-Environment User's Guide, version 26.
Centrum voor Wiskunde en Informatica (CWI), February 1993.
Available by *ftp* from `ftp.cwi.nl:pub/gipe/reports` as `SysManual.ps.Z`.

[37] (17,II.1) P. Klint.
A meta-environment for generating programming environments.
ACM Transactions on Software Engineering and Methodology, 2(2):176–201, 1993.

[38] (1) Donald E. Knuth.
The Art of Computer Programming.
Addison-Wesley, 1968.

[39] (55) Donald E. Knuth.
Semantics of context-free languages.
Mathematical Systems Theory, 2(2):127–145, June 1968.
See also [40].

[40] (55,I.6) Donald E. Knuth.
Semantics of context-free languages

(correction).
Mathematical Systems Theory, 5(1):95–96, May 1971.

[41] (4,38) Pieter Koopman. *Functional Programs as Executable Specifications.* PhD thesis, University of Nijmegen, Toernooiveld 1, Nijmegen, The Netherlands, 1990.

[42] (69) C. H. A. Koster. Affix grammars. In *Algol68 Implementation.* North Holland Publishing Company, Utrecht, The Netherlands, 1971.

[43] (1) C.H.A. Koster. Using the CDL compiler compiler. In F. L. Bauer and J. Eickel, editors, *Compiler Construction, An Advanced Course*, volume 21 of *Lecture Notes in Computer Science*. Springer-Verlag, 1974.

[44] (55) P. Kühling. *Affix-Grammatiken zur Beschreibung von Programmiersprachen.* PhD thesis, Technische Universität Berlin, Fachbereich Informatik, February 1978. In German.

[45] (40) Wil Lamain. Update plans, implementatie aspecten. Master's thesis, University of Nijmegen, Toernooiveld 1, Nijmegen, The Netherlands, 1992. (In Dutch).

[46] (56) B.M. Leavenworth. Syntax macros and extended translation. *Communications of the ACM*, 9(11):790–793, November 1966.

[47] (vii,4) Hendrik C.R. Lock. An abstract machine for the implementation of functional logic programming languages. Technical report, ESPRIT Basic Research Action No. 3147 (the Phœnix Project), 1990.

[48] (vii) Hendrik C.R. Lock.
A specification of the JCode instructions for the JUMP machine, 1991.
Personal communication.

[49] (55) Brian H. Mayoh.
Attribute grammmars and mathematical semantics.
SIAM Journal on Computing, 10(3):503–518, August 1981.

[50] (1) E. Meijer, M.M. Fokkinga, and R. Paterson.
Functional programming with bananas, lenses, envelopes and barbed wire.
In John Hughes, editor, *Functional Programming and Computer Architecture*, volume 523 of *Lecture Notes in Computer Science*. Springer-Verlag, 1991.

[51] (1,116) Erik Meijer.
Calculating Compilers.
PhD thesis, University of Nijmegen, Toernooiveld 1, Nijmegen, The Netherlands, 1992.

[52] (vii,25,29) Erik Meijer and Ross Patterson.
Down with lambda lifting!
Technical report, ESPRIT Basic Research Action No. 3147 (the Phœnix Project), 1991.

[53] (vii,1,2,3,43,69,111,115) Hans Meijer.
Programmar: A Translator Generator.
PhD thesis, University of Nijmegen, Toernooiveld 1, Nijmegen, The Netherlands, 1986.

[54] (1) P. Naur, (editor).
Revised report on the algorithmic language algol 60.
Communications of the ACM, 6:1–17, 1963.

[55] (69) Hugh Osborne.
Choosy affix evaluation for EAG parsers.
Master's thesis, University of Nijmegen, Toernooiveld 1, Nijmegen, The Netherlands, 1989.

[56] Hugh Osborne.
(vii) Update plans.
Periodica Polytechnica Ser. El. Eng., 35(3):153–163, 1991.

[57] Hugh Osborne.
(vii) Update plans, an example: Flip.
Technical report, ESPRIT Basic Research Action No. 3147 (the Phœnix Project), 1991.

[58] Hugh Osborne.
(vii) Update plans, an introduction.
Technical report, ESPRIT Basic Research Action No. 3147 (the Phœnix Project), 1991.

[59] Hugh Osborne.
(vii) The semantics and syntax of update schemes.
In *Code Generation — Concepts, Tools, Techniques (Proceedings of the International Workshop on Code Generation, Dagstuhl, Germany, 20–24 May 1991)*, Workshops in Computing. Springer Verlag, 1992.

[60] Hugh Osborne.
(vii) Update plans.
In *Proceedings of the 25th Hawaii International Conference on System Sciences.* IEEE Computer Society Press, 1992.

[61] Rohit J. Parikh.
(*stellingen*) On context-free languages.
Journal of the Association for Computing Machinery, 13(4):570–581, October 1966.

[62] H. Partsch.
(1) *Specification and Transformation of Programs - a Formal Approach to Software Development.*
Springer-Verlag, Berlin, 1990.

[63] Simon L. Peyton Jones and Jon Salkild.
(4) The spineless tagless G-machine.
University College, London, 1989.

[64] D.L. Pilling.
(*stellingen*) Commutative regular equations and Parikh's

theorem.
Journal of the London Mathematical Society, 2(6):663–666, 1973.

[65] (41) G.D. Plotkin.
A powerdomain construction.
SIAM Journal on Computing, 5(3):452–487, 1976.

[66] (vii) Sara and Tobias Osborne.
Personal communication.

[67] (117) Janos Sarbo.
Collegedictaat vertalerbouw.
Technical report, University of Nijmegen, Toernooiveld 1, Nijmegen, The Netherlands, 1991.
(In Dutch).

[68] (4) Daniel P. Siewiorek, C. Gordon Bell, and Allen Newell.
Computer Structures: Principles and Examples.
Computer Science Series. McGraw-Hill, 1982.

[69] (1) C.P. Snow.
The Two Cultures and the Scientific Revolution.
Cambridge University Press, 1959.

[70] (1) STOP.
STOP International Summer School on Constructive Algorithmics, Ameland, September 1989.
Lecture notes.

[71] (1) STOP.
Lecture Notes of the STOP 1992 Summer School on Constructive Algorithmics, Ameland, 1992.

[72] (2,I.2) Marko van Eekelen.
Parallel Graph Rewriting.
PhD thesis, University of Nijmegen, Toernooiveld 1, Nijmegen, The Netherlands, 1988.

[73] (4) D. H. D. Warren.
An abstract Prolog instruction set.

Technical Report 309, Artificial Intelligence Center, SRI International, Menlo Park, California, USA, 1983.

[74] (69) D. A. Watt. *Analysis Oriented Two Level Grammars.* PhD thesis, University of Glasgow, 1974.

[75] (1) A. van Wijngaarden, B.J. Mailloux, J.E.L. Peck, C.H.A. Koster, M. Sintzoff, C.H. Lindsey, L.G.L.T. Meertens, and R.G. Fisker, editors. *Revised Report on the Algorithmic Language* ALGOL 68. Springer-Verlag, 1976.

Appendix II Grammar

This appendix gives a full context free grammar for Update Plans as specified in this thesis. The grammar is given as an ASF+SDF [36, 37] specification, a specification formalism developed at the University of Amsterdam and the *Centrum voor Wiskunde en Informatica*. The specification can be read as a context free grammar, in which production rules read from right to left. The following notational conventions apply:

- $S*$ defines zero or more repetitions of sort S;
- $S+$ defines one or more repetitions of sort S;
- $\{S\ sep\}*$ defines zero or more repetitions of sort S separated by the literal *sep*;
- $\{S\ sep\}+$ defines one or more repetitions of sort S separated by the literal *sep*;
- $[s]$ represents any one of the characters in the string s;
- $\sim[s]$ represents any character not in the string s;
- `\t` is the horizontal tabulation character;
- `\n` is the newline character.

The {**left**} and {**bracket**} annotation is applied to disambiguate parsings, as is the information provided under a **priorities** header.

module Plans

imports Layout Alternatives Archetypes ParallelBlocks Stores Types

exports

sorts SCRIPT PLAN ITEM

context-free functions

CONFIGURATION "." PLAN → SCRIPT

ITEM* → PLAN

ALTERNATIVES → ITEM
ARCHETYPE-DEFINITION → ITEM
PARBLOCK → ITEM
STORE-DECLARATION → ITEM
TYPE-DECLARATION → ITEM

module ParallelBlocks

imports Alternatives

exports

sorts PARBLOCK

context-free functions

"(||" ALTERNATIVES* "||)" → PARBLOCK

module Alternatives

imports Schemes

exports

sorts ALTERNATIVES

context-free functions

{SCHEME ";"}+ "." → ALTERNATIVES

module Archetypes

imports BasicArchetypes CommandArchetypes Terms

hiddens

sorts INDEX-OPT

exports

sorts ARCHETYPE-DEFINITION ARCHETYPE-CALL PARAMETERS

context-free functions

COMMAND-ARCHETYPE-DEFINITION → ARCHETYPE-DEFINITION
BASIC-ARCHETYPE-DEFINITION → ARCHETYPE-DEFINITION

ARCHETYPE-CALL → TERM

ARCHETYPE-NAME INDEX-OPT PARAMETERS → ARCHETYPE-CALL

"(" {TERM ","}* ")" → PARAMETERS

INDEX → INDEX-OPT
→ INDEX-OPT

In the production rules for update schemes layout is forbidden between a locator and the '[' or ']' of the cell sequence of which it is a locator.

module Schemes

imports Terms

hiddens

sorts LOC-EXPR LEFT-SECTION LOCATOR

exports

sorts SCHEME CONFIGURATION TEXT CONTEXT REPEAT GUARD

context-free functions

CONFIGURATION GUARD CONFIGURATION → SCHEME

REPEAT GUARD CONFIGURATION → SCHEME

TEXT CONTEXT → CONFIGURATION

TERM * → TEXT

LOC-EXPR * → CONTEXT

"|" → REPEAT

"= [" TERM "] =>" → GUARD

"==>" → GUARD

LEFT-SECTION+ LOCATOR → LOC-EXPR

"?" TERM-OPT "[" TEXT "]" → LOC-EXPR

"!" TERM-OPT "[" TEXT "]" → LOC-EXPR

LOCATOR "[" TEXT "]" → LEFT-SECTION

TERM-OPT → LOCATOR

module BasicArchetypes

imports Terms Schemes

hiddens

sorts BASIC-DECLARATION BASIC-DEFINITION BASIC-BODY

exports

sorts BASIC-ARCHETYPE-DEFINITION

context-free functions

BASIC-DECLARATION BASIC-DEFINITION+ → BASIC-ARCHETYPE-DEFINITION

BASIC-ARCHETYPE-NAME PARAMETERS → BASIC-DECLARATION

"=" BASIC-BODY "." → BASIC-DEFINITION

CONFIGURATION GUARD CONFIGURATION → BASIC-BODY
REPEAT GUARD CONFIGURATION → BASIC-BODY
CONFIGURATION → BASIC-BODY

TEXT "|" CONTEXT GUARD CONTEXT → BASIC-BODY
TEXT "|" REPEAT GUARD CONTEXT → BASIC-BODY
TEXT "|" CONTEXT → BASIC-BODY

module CommandArchetypes

imports Terms Schemes

hiddens

sorts COMMAND-DECLARATION COMMAND-DEFINITION COMMAND-BODY

exports

sorts COMMAND-ARCHETYPE-DEFINITION

context-free functions

COMMAND-DECLARATION COMMAND-DEFINITION+ → COMMAND-ARCHETYPE-DEFINITION

COMMAND-ARCHETYPE-NAME PARAMETERS TEXT → COMMAND-DECLARATION

"=" COMMAND-BODY "." → COMMAND-DEFINITION

CONTEXT GUARD CONFIGURATION → COMMAND-BODY
REPEAT GUARD CONFIGURATION → COMMAND-BODY
CONTEXT → COMMAND-BODY

module Stores

imports Lexicon

exports

sorts STORE STORE-STRUCTURE STORE-DECLARATION

context-free functions

STORE-NAME	→	STORE-STRUCTURE
"(" STORE-STRUCTURE ")"	→	STORE-STRUCTURE {**bracket**}
STORE-STRUCTURE "++" STORE-STRUCTURE	→	STORE-STRUCTURE {**left**}
STORE-STRUCTURE "|" STORE-STRUCTURE	→	STORE-STRUCTURE {**left**}
STORE-STRUCTURE "*"	→	STORE-STRUCTURE
"{" {STORE ","}* "}"	→	STORE-DECLARATION
STORE-NAME	→	STORE
STORE-NAME "=" STORE-STRUCTURE	→	STORE

priorities

STORE-STRUCTURE "*" → STORE-STRUCTURE >

STORE-STRUCTURE "++" STORE-STRUCTURE → STORE-STRUCTURE >

STORE-STRUCTURE "|" STORE-STRUCTURE → STORE-STRUCTURE

module Types

imports Terms Stores

exports

sorts TYPE-DECLARATION

context-free functions

{TERM ","}+ "::" STORE-STRUCTURE "."	→	TYPE-DECLARATION

module Terms

imports BasicTerms Arithmetic Logic Ordering Memory Stores

exports

sorts TERM-OPT

context-free functions

"(" TERM "::" STORE-STRUCTURE ")"	→ TERM
TERM "*"	→ TERM
TERM "++" TERM	→ TERM {**left**}
TERM "|" TERM	→ TERM {**left**}
"(" TERM ")"	→ TERM {**bracket**}
TERM	→ TERM-OPT
	→ TERM-OPT

priorities

TERM "*" → TERM > TERM "++" TERM → TERM > TERM "|" TERM → TERM

module BasicTerms

imports Lexicon

hiddens

sorts

exports

sorts TERM

context-free functions

VARIABLE	→ TERM
NUMBER	→ TERM
CHAR	→ TERM
SYMB-CONST	→ TERM
DONTCARE	→ TERM

module Arithmetic

imports BasicTerms

exports

context-free functions

TERM "+" TERM → TERM {**left**}
TERM "−" TERM → TERM {**left**}
TERM "×" TERM → TERM {**left**}
TERM "/" TERM → TERM {**left**}
TERM "^" TERM → TERM {**left**}
"−" TERM → TERM

priorities

"−" TERM → TERM > "^" > { "×", "/" } >
{ "+", TERM "−" TERM → TERM }

module Logic

imports BasicTerms

exports

context-free functions

TERM "∧" TERM → TERM {**left**}
TERM "∨" TERM → TERM {**left**}
"¬" TERM → TERM

priorities

"¬" TERM → TERM > "∧" > "∨"

module Ordering

imports BasicTerms

exports

context-free functions

TERM "<" TERM → TERM
TERM "≤" TERM → TERM
TERM "=" TERM → TERM
TERM "≠" TERM → TERM
TERM "≥" TERM → TERM
TERM ">" TERM → TERM

module Memory

imports Lexicon BasicTerms

exports

context-free functions

"|" STORE-NAME "|" → TERM
"@" "(" TERM "," TERM ")" → TERM
"@" TERM → TERM

module Lexicon

exports

sorts VARIABLE NUMBER CHAR SYMB-CONST DONTCARE
STORE-NAME
BASIC-ARCHETYPE-NAME COMMAND-ARCHETYPE-NAME ARCHETYPE-NAME
INDEX

lexical syntax

[0-9]+	→ NUMBER
"'" ~['] "'"	→ CHAR
[A-Z] [A-Z'0-9_]*	→ SYMB-CONST
[a-z] [a-z'0-9_]*	→ VARIABLE
"_"	→ DONTCARE
[A-Z] [A-Z0-9_']* [a-z] [A-Za-z0-9_']*	→ STORE-NAME
[a-z] [a-z'0-9_]*	→ BASIC-ARCHETYPE-NAME
[A-Z] [A-Z'0-9_]*	→ COMMAND-ARCHETYPE-NAME
BASIC-ARCHETYPE-NAME	→ ARCHETYPE-NAME
COMMAND-ARCHETYPE-NAME	→ ARCHETYPE-NAME
NUMBER	→ INDEX
"{" NUMBER "}"	→ INDEX

module Layout

exports

lexical syntax

[\t\n]*	→ LAYOUT
"||"	→ LAYOUT

```
[2.1] NEW_ENV n HEAP[h] ENV[e] e[env]e+n
      ⟹ ENV[h] h[env]i HEAP[i].
[2.2] BIND i r r[h] ENV[e] ⟹ e+i[h].
[2.3] LOOKUP i r ENV[e] e+i[h] ⟹ r[h].
[2.4] PUSH r r[h] SP[t] ⟹ SP[s] s[h]t.
[2.5a] POP r SP[s] TP[u] u[w] s[h]t
      =[s<w]⇒ r[h] SP[t].
[2.5b] POP PC[p] SP[s] TP[u] u[s]v s[h]t
                          h[SUSP c e] ENV[f] HEAP[i]
      ⟹ PC[c] ENV[e] TP[v]
                          i[SUSP p f]j HEAP[j] s[i].
[2.6] SUSP r c ENV[e] HEAP[h]
      ⟹ r[h] h[SUSP c e]i HEAP[i].
[2.7a] JUMP r r[h] h[SUSP c e] ⟹ PC[c] ENV[e].
[2.7b] JUMP r r[h] h[CONST] TP[u]
                             u[s]v s[i]t i[SUSP c e]
      ⟹ PC[c] ENV[e] TP[v] s[h].
[3.1] ASSIGN r n ⟹ r[n].
[3.2] CONST r r[n] HEAP[h]
      ⟹ h[CONST n]i r[h] HEAP[i].
```

[4.1a] CALL r r[h] h[CONST] SP[t] $\Longrightarrow$ SP[s] s[h]t.

[4.1b] PC[p] p[CALL r]q r[h] h[SUSP c e]
ENV[f] HEAP[i] SP[t] TP[v]
$\Longrightarrow$ PC[c] ENV[e] i[SUSP q f]j
HEAP[j] s[i]t SP[s] TP[u] u[s]v.

[5.1] ADD r1 r0 r1[h] h[CONST x] r0[i] i[CONST y]
$\Longrightarrow$r0[x+y].

[6.1] READ r ?[n] $\Longrightarrow$ r[n].

[6.2] WRITE r r[h] h[CONST n] $\Longrightarrow$![n].

Explanation of figure 1

1. If there is neither a read nor a write access the store can, for all intents and purposes, be said to not exist. However, no-access memories can appear as part of a configuration resulting from a transformation between configurations (see chapter eight).

2. Nothing is read from the store, and output is written sequentially, either right to left, or left to right.

3. This could be some sort of storage device, having data written to it. The data may be used in some other context, so that in the update plan under consideration it functions as "write-only memory".

4. See output stream.

5. This case is further analysed in figure 2.

6. There seems to be no standard type of addressing structure corresponding to this combination.

	write		
read	no such access	structured access	unstructured access
no such access	no-access memory (1)	output stream (2)	write only memory (3)
structured access	input stream (4)	— (5)	??? (6)
unstructured access	read only memory (7)	heap (8)	general memory (9)

Figure 1: Access structure and store structure.

		write	
		left	right
read	left	queue(a)	stack (b_1)
	right	stack (b_2)	queue(a)

Figure 2: Structured read/write accesses.

7. For example, a programme store.

8. This case only has a real heap-like structure if an extra condition is guaranteed, namely that, at all times, any read address is in the luff of the write dedicated register. This is true if it can be shown that every locator in the store that is created, rather than copied, is, at the time of its creation, in the luff of the write dedicated register.

9. No structure can be detected.

Explanation of figure 2

a) Two dedicated registers — one read register, one write register

b) One register for read *and* write accesses, in case b_1 the stack grows to the right, in b_2 to the left.

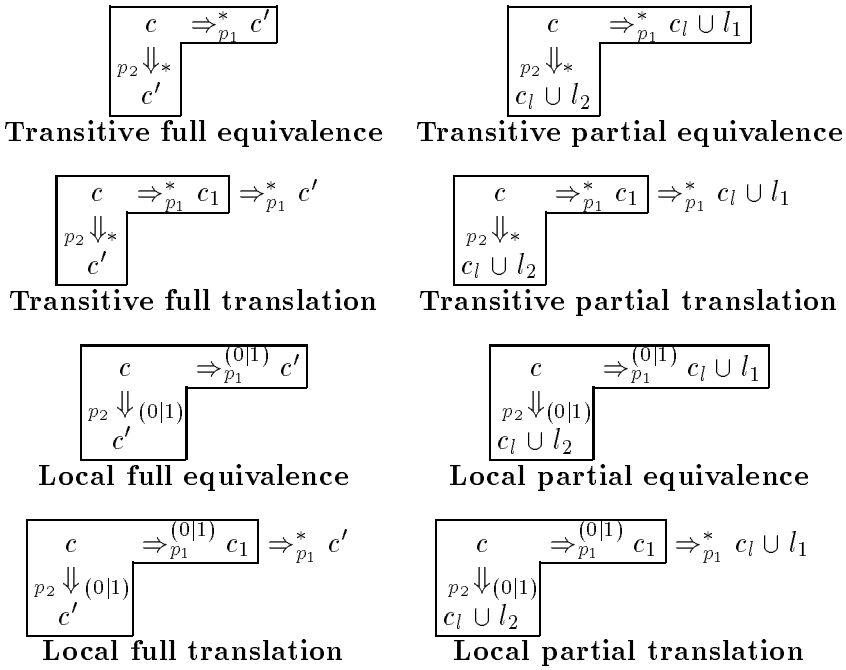


Figure 1: Equivalence and translation properties

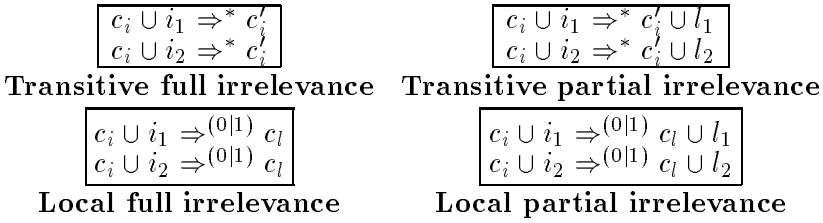


Figure 2: Irrelevance properties

$$\texttt{MOV rop(x) wop(a,\_)} \implies \texttt{a[x]}.$$
$$\texttt{MEA adr(x) wop(a,\_)} \implies \texttt{a[x]}.$$

$$\texttt{arith(x,y,r) rop(x) wop(a,y)} \implies \texttt{a[r]}.$$

$$\texttt{bool(x,y,r) rop}_1\texttt{(x) rop}_2\texttt{(y)} \implies \texttt{CC[r]}.$$

$$\texttt{PC[cp] pc[jump(cond) trg(t)]qc} \;=[\; \texttt{cond} \;]\Rightarrow\; \texttt{PC[t]}.$$
$$=[\; \neg\texttt{(cond)} \;]\Rightarrow\; \texttt{PC[qc]}.$$

$$\texttt{PC[pc] pc[JSR trg(t)]qc SP[tp]} \Longrightarrow \texttt{PC[t] SP[sp] sp[qc]tp}.$$

$$\texttt{PC[pc] pc[RET] SP[sp] sp[pc]tp} \Longrightarrow \texttt{PC[pc] SP[tp]}.$$

Figure 1: Tree and linear machine update schemes.

$$
\begin{array}{l}
\mathtt{REG(r,b)\ r} \quad\ = \mathtt{r[b].} \\
\mathtt{REGDEF(b,v)\ r} \quad = \mathtt{r[b]\ b[v].} \\
\mathtt{BDISP(b{+}d,v)\ r\ d} = \mathtt{r[b]\ b{+}d[v].} \\
\mathtt{BDISPDEF(a,v)\ r\ d} = \mathtt{r[b]\ b{+}d[a]\ a[v].} \\
\mathtt{PREDEC(a,v)\ r} \quad = \mathtt{r[b]\ a[v]b} \quad\ \Longrightarrow \mathtt{r[a].} \\
\mathtt{POSTINC(b,v)\ r} \quad = \mathtt{r[b]\ b[v]c} \quad\ \Longrightarrow \mathtt{r[c].}
\end{array}
$$

$$
\begin{array}{lll}
\mathtt{wop(a,v)} = \mathtt{REG(a,v).} & \mathtt{rop(v)} = \mathtt{REG(\_,v).} \\
\qquad\ = \mathtt{REGDEF(a,v).} & \qquad\ = \mathtt{REGDEF(\_,v).} \\
\qquad\ = \mathtt{BDISP(a,v).} & \qquad\ = \mathtt{BDISP(\_,v).} \\
\qquad\ = \mathtt{BDISPDEF(a,v).} & \qquad\ = \mathtt{BDISPDEF(\_,v).} \\
\qquad\ = \mathtt{PREDEC(a,v).} & \qquad\ = \mathtt{POSTINC(\_,v).} \\
 & \qquad\ = \mathtt{IMM\ v.} \\
\mathtt{adr(a)} = \mathtt{REGDEF(a,\_).} \\
\qquad = \mathtt{BDISP(a,\_).} & \mathtt{trg(t)} = \mathtt{adr(t).} \\
\qquad = \mathtt{BDISPDEF(a,\_).} & \qquad\ = \mathtt{LAB\ t.}
\end{array}
$$

$$
\begin{array}{lll}
\mathtt{arith(x,y,y{+}x)} \ = \mathtt{ADD.} & & \mathtt{bool(x,y,x<y)} = \mathtt{CLT.} \\
\mathtt{arith(x,y,y-x)} \ = \mathtt{SUB.} & & \mathtt{bool(x,y,x\le y)} = \mathtt{CLE.} \\
\mathtt{arith(x,y,y*x)} \ = \mathtt{MUL.} & & \mathtt{bool(x,y,x=y)} = \mathtt{CEQ.} \\
\mathtt{arith(x,y,y/x)} \ = \mathtt{DIV} \Rightarrow\!\!\lfloor\ \mathtt{x\neq 0}\ \rfloor\!\!\Rightarrow . & & \mathtt{bool(x,y,x\neq y)} = \mathtt{CNE.} \\
 & & \mathtt{bool(x,y,x\ge y)} = \mathtt{CGE.} \\
\mathtt{jump(cc)} \quad = \mathtt{JT\ CC[cc].} & & \mathtt{bool(x,y,x>y)} = \mathtt{CGT.} \\
\mathtt{jump(\neg(cc))} = \mathtt{JF\ CC[cc].} \\
\mathtt{jump(TRUE)} \ = \mathtt{JMP.}
\end{array}
$$

Figure 2: Linear machine archetypes.

```
REG(r,b) r              = r[b].
REGDEF(b,v) r           = r[b] b[v].
BDISP(b+d,v) r d        = r[b] b+d[v].
BDISPDEF(a,v) r d       = r[b] b+d[a] a[v].
PREDEC(a,v) r           = r[b] a[v]b       ⟹r[a].
POSTINC(b,v) r          = r[b] b[v]c       ⟹r[c].
IREG(_,b) ir(b)         = .
IREGDEF(b,v) ir(b)      = b[v].
IBDISP(b+d,v) ir(b) d   = b+d[v].
IBDISPDEF(a,v) ir(b) d  = b+d[a] a[v].

kern(a,v) = REGDEF(a,v).    rop(v) = kern(_,v).
          = BDISP(a,v).             = IMM v.
          = BDISPDEF(a,v).          = REG(_,v).
          = IBDISP(a,v).            = POSTINC(_,v).
          = IBDISPDEF(a,v).         = IREG(_,v).
                                    = IREGDEF(_,v).

wop(a,v) = kern(a,v).       adr(a) = kern(a,_).
         = REG(a,v).        trg(t) = LAB t.
         = PREDEC(a,v).     tmp(t) = IREG(_,t).
         = IREGDEF(a,v).

iarith(y+x)  = IADD rop(x) tmp(y).
iarith(y−x)  = ISUB rop(x) tmp(y).
iarith(y∗x)  = IMUL rop(x) tmp(y).
iarith(y/x)  = IDIV rop(x) tmp(y) =[ x ≠ 0 ]⇛ .

imov(x) = IMOV rop(x) (IREG NIL).  ir(r) = iarith(r).
imov(a) = IMEA adr(a) (IREG NIL).        = imov(r).

arith(x,y,y+x)  = ADD.              bool(x,y,x<y) = CLT.
arith(x,y,y−x)  = SUB.              bool(x,y,x≤y) = CLE.
arith(x,y,y∗x)  = MUL.              bool(x,y,x=y) = CEQ.
arith(x,y,y/x)  = DIV =[ x ≠ 0 ]⇛ . bool(x,y,x≠y) = CNE.
                                    bool(x,y,x≥y) = CGE.
jump(cc)    = JT CC[cc].            bool(x,y,x>y) = CGT.
jump(¬(cc)) = JF CC[cc].
jump(TRUE)  = JMP.
```

Figure 3: Tree machine archetypes.

$\{1.5\}$
$=$ $\texttt{arith(x,y,r) rop(x) wop(a,y)}$
$\texttt{==>}$ $\texttt{a[r].}$
$\{1.4$ $\texttt{arith(x,y,y - x) = SUB.}\}$
$=$ $\texttt{SUB rop(x) wop(a,y)}$ $r = y - x$
$\texttt{==>}$ $\texttt{a[r].}$
$\{1.2$ $\texttt{rop}(\texttt{v}_1) = \texttt{POSTINC}(\_, \texttt{v}_1).\}$
$=$ $\texttt{SUB POSTINC}(\_, \texttt{v}_1)\ \texttt{wop(a,y)}$ $x = v_1$
$\texttt{==>}$ $\texttt{a[r].}$
$\{1.1$ $\texttt{POSTINC}(\texttt{b}_1, \texttt{v}_2)\ \texttt{r}_1 = \texttt{r}_1[\texttt{b}_1]\ \texttt{b}_1[\texttt{v}_2]\texttt{c}_1 \texttt{==>} \texttt{r}_1[\texttt{c}_1].\}$
$=$ $\texttt{SUB (POSTINC r}_1)\ \texttt{wop(a,y)}$ $\_ = b_1, v_1 = v_2$
$\texttt{r}_1[\texttt{b}_1]\ \texttt{b}_1[\texttt{v}_2]\texttt{c}_1$
$\texttt{==>r}_1[\texttt{c}_1]$ $\texttt{a[r].}$
$\{1.2$ $\texttt{wop}(\texttt{a}_3, \texttt{v}_3) = \texttt{REGDEF}(\texttt{a}_3, \texttt{v}_3).\}$
$=$ $\texttt{SUB (POSTINC r}_1)\ \texttt{REGDEF}(\texttt{a}_3, \texttt{v}_3)$ $a = a_3, y = v_3$
$\texttt{r}_1[\texttt{b}_1]\ \texttt{b}_1[\texttt{v}_2]\texttt{c}_1$
$\texttt{==>r}_1[\texttt{c}_1]$ $\texttt{a[r].}$
$\{1.1$ $\texttt{REGDEF}(\texttt{b}_4, \texttt{v}_4)\ \texttt{r}_2 = \texttt{r}_2[\texttt{b}_4]\ \texttt{b}_4[\texttt{v}_4].\}$
$=$ $\texttt{SUB (POSTINC r}_1)\ \texttt{(REGDEF r}_2)$ $a_3 = b_4, v_3 = v_4$
$\texttt{r}_1[\texttt{b}_1]\ \texttt{b}_1[\texttt{v}_2]\texttt{c}_1\ \texttt{r}_2[\texttt{b}_4]\ \texttt{b}_4[\texttt{v}_4]$
$\texttt{==>r}_1[\texttt{c}_1]$ $\texttt{a[r].}$

Figure 4: Archetype expansion in the linear machine.

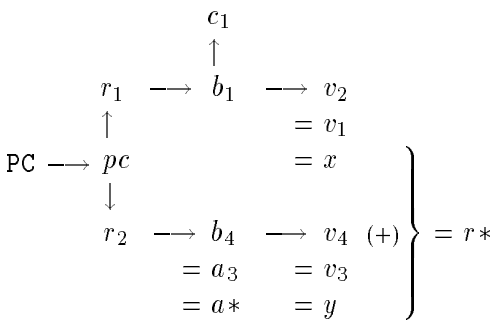


Figure 5: Parameter resolution for the archetype expansion in figure 4

$\texttt{SUB (POSTINC r}_1)\ \texttt{(REGDEF r}_2)$
$\texttt{r}_1[\texttt{b}_1]\ \texttt{b}_1[\texttt{v}_2]\texttt{c}_1\ \texttt{r}_2[\texttt{b}_4]\ \texttt{b}_4[\texttt{v}_4]$
$\texttt{==>r}_1[\texttt{c}_1]$ $\texttt{b}_4[\texttt{v}_4 - \texttt{v}_2].$

Figure 6: The expanded update scheme from figures 4 and 5

$$\texttt{MOV rop(x) wop(a,\_)} \Longrightarrow \texttt{a[x]}.$$
$$\begin{bmatrix}\texttt{rop}(\texttt{v}_1) = \texttt{kern}(\texttt{\_}_1, \texttt{v}_1)\\ \texttt{wop}(\texttt{a}_2, \texttt{v}_2) = \texttt{IREGDEF}(\texttt{a}_2, \texttt{v}_2)\end{bmatrix} \qquad x = v_1,\ a = a_2,\ \_ = v_2$$

$$\texttt{MOV kern}(\texttt{\_}_1, \texttt{v}_1)\ \texttt{IREGDEF}(\texttt{a}_2, \texttt{v}_2) \Longrightarrow \texttt{a[x]}.$$
$$\begin{bmatrix}\texttt{kern}(\texttt{a}_3, \texttt{v}_3) = \texttt{IBDISP}(\texttt{a}_3, \texttt{v}_3)\\ \texttt{IREGDEF}(\texttt{a}_4, \texttt{v}_4)\ \texttt{ir}(\texttt{a}_4) = \texttt{a}_4\texttt{[v}_4\texttt{]}\end{bmatrix} \qquad \texttt{\_}_1 = a_3,\ v_1 = v_3,\ a_2 = a_4,\ v_2 = v_4$$

$$\texttt{MOV IBDISP}(\texttt{a}_3, \texttt{v}_3)\ (\texttt{IREGDEF ir}(\texttt{a}_4))\ \ \texttt{a}_4\texttt{[v}_4\texttt{]} \Longrightarrow \texttt{a[x]}.$$
$$\begin{bmatrix}\texttt{IBDISP}(\texttt{b}_5\texttt{+d}_5, \texttt{v}_5)\ \texttt{ir}(\texttt{b}_5)\ \texttt{d}_5 = \texttt{b}_5\texttt{+d}_5\texttt{[v}_5\texttt{]}\\ \texttt{ir}(\texttt{r}_6) = \texttt{imov}(\texttt{r}_6)\end{bmatrix} \qquad a_3 = b_5 + d_5,\ v_3 = v_5,\ a_4 = r_6$$

$$\texttt{MOV (IBDISP ir}(\texttt{b}_5)\ \texttt{d}_5)\ (\texttt{IREGDEF imov}(\texttt{r}_6))\ \ \texttt{b}_5\texttt{+d}_5\texttt{[v}_5\texttt{]}\ \ \texttt{a}_4\texttt{[v}_4\texttt{]} \Longrightarrow \texttt{a[x]}.$$
$$\begin{bmatrix}\texttt{ir}(\texttt{r}_7) = \texttt{imov}(\texttt{r}_7)\\ \texttt{imov}(\texttt{x}_8) = \texttt{IMOV rop}(\texttt{x}_8)\ (\texttt{IREG NIL})\end{bmatrix} \qquad b_5 = r_7,\ r_6 = x_8$$

$$\texttt{MOV (IBDISP imov}(\texttt{r}_7)\ \texttt{d}_5)\ (\texttt{IREGDEF (IMOV rop}(\texttt{x}_8)\ (\texttt{IREG NIL})))\ \ \texttt{b}_5\texttt{+d}_5\texttt{[v}_5\texttt{]}\ \ \texttt{a}_4\texttt{[v}_4\texttt{]} \Longrightarrow \texttt{a[x]}.$$
$$\begin{bmatrix}\texttt{imov}(\texttt{x}_9) = \texttt{IMOV rop}(\texttt{x}_9)\ (\texttt{IREG NIL})\\ \texttt{rop}(\texttt{v}_{10}) = \texttt{IMM v}_{10}\end{bmatrix} \qquad r_7 = x_9,\ x_8 = v_{10}$$

$$\texttt{MOV (IBDISP (IMOV rop}(\texttt{x}_9)\ (\texttt{IREG NIL}))\ \texttt{d}_5)\ (\texttt{IREGDEF (IMOV (IMM v}_{10})\ (\texttt{IREG NIL})))\ \ \texttt{b}_5\texttt{+d}_5\texttt{[v}_5\texttt{]}\ \ \texttt{a}_4\texttt{[v}_4\texttt{]} \Longrightarrow \texttt{a[x]}.$$
$$\left[\texttt{rop}(\texttt{v}_{11}) = \texttt{kern}(\texttt{\_}_{11}, \texttt{v}_{11})\right] \qquad x_9 = v_{11}$$

$$\texttt{MOV (IBDISP (IMOV kern}(\texttt{\_}_{11}, \texttt{v}_{11})\ (\texttt{IREG NIL}))\ \texttt{d}_5)\ (\texttt{IREGDEF (IMOV (IMM v}_{10})\ (\texttt{IREG NIL})))\ \ \texttt{b}_5\texttt{+d}_5\texttt{[v}_5\texttt{]}\ \ \texttt{a}_4\texttt{[v}_4\texttt{]} \Longrightarrow \texttt{a[x]}.$$
$$\left[\texttt{kern}(\texttt{a}_{12}, \texttt{v}_{12}) = \texttt{BDISP}(\texttt{a}_{12}, \texttt{v}_{12})\right] \qquad \texttt{\_}_{11} = a_{12},\ v_{11} = v_{12}$$

$$\texttt{MOV (IBDISP (IMOV BDISP}(\texttt{a}_{12}, \texttt{v}_{12})\ (\texttt{IREG NIL}))\ \texttt{d}_5)\ (\texttt{IREGDEF (IMOV (IMM v}_{10})\ (\texttt{IREG NIL})))\ \ \texttt{b}_5\texttt{+d}_5\texttt{[v}_5\texttt{]}\ \ \texttt{a}_4\texttt{[v}_4\texttt{]} \Longrightarrow \texttt{a[x]}.$$
$$\left[\texttt{BDISP}(\texttt{b}_{13}\texttt{+d}_{13}, \texttt{v}_{13})\ \texttt{r}_{13}\ \texttt{d}_{13} = \texttt{r}_{13}\texttt{[b}_{13}\texttt{]}\ \texttt{b}_{13}\texttt{+d}_{13}\texttt{[v}_{13}\texttt{]}\right] \qquad a_{12} = b_{13} + d_{13},\ v_{12} = v_{13}$$

$$\texttt{MOV (IBDISP (IMOV (BDISP r}_{13}\ \texttt{d}_{13})\ (\texttt{IREG NIL}))\ \texttt{d}_5)\ (\texttt{IREGDEF (IMOV (IMM v}_{10})\ (\texttt{IREG NIL})))\ \ \texttt{r}_{13}\texttt{[b}_{13}\texttt{]}\ \texttt{b}_{13}\texttt{+d}_{13}\texttt{[v}_{13}\texttt{]}\ \texttt{b}_5\texttt{+d}_5\texttt{[v}_5\texttt{]}\ \ \texttt{a}_4\texttt{[v}_4\texttt{]} \Longrightarrow \texttt{a[x]}.$$

Figure 7: Textual expansion of archetypes in the tree machine

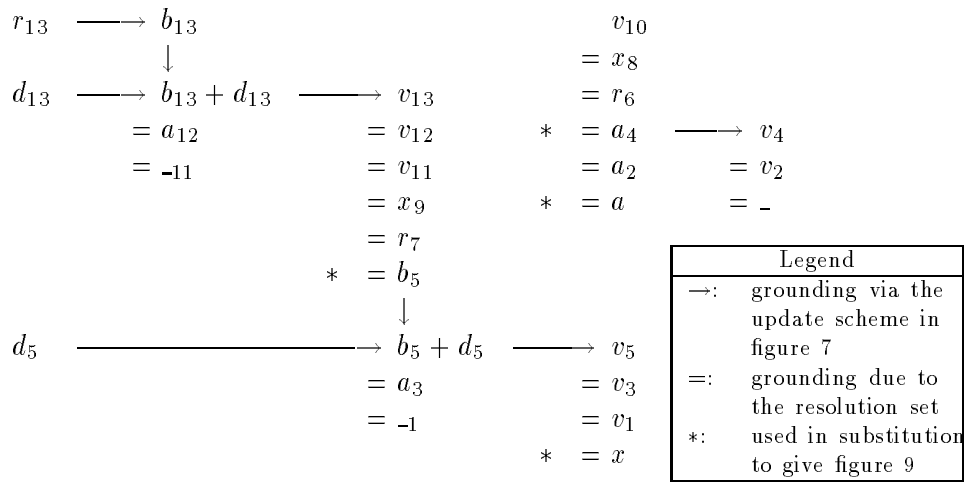


Figure 8: Parameter resolution for the archetype expansion in figure 7

```
MOV (IBDISP (IMOV (BDISP r13 d13) (IREG NIL)) d5)
    (IREGDEF (IMOV (IMM v10) (IREG NIL)))
    r13[b13] b13+d13[v13] v13+d5[v5]        v10[v4]==>v10[v5].
```

Figure 9: The expanded update scheme from figures 7 and 8

$$
\begin{aligned}
& \mathcal{T}\left[\!\!\left[\begin{array}{l}\texttt{MOV (IBDISP (IMOV (BDISP r}_{13}\texttt{ d}_{13}\texttt{) (IREG NIL)) d}_5\texttt{)}\\ \quad\texttt{(IREGDEF (IMOV (IMM v}_{10}\texttt{) (IREG NIL)))}\end{array}\right]\!\!\right] n \\
= \;& \mathcal{IR}[\![\texttt{IREGDEF (IMOV (IMM v}_{10}\texttt{) (IREG NIL))}]\!]\ n \\
& \mathcal{IR}[\![\texttt{IBDISP (IMOV (BDISP r}_{13}\texttt{ d}_{13}\texttt{) (IREG NIL)) d}_5]\!]\ n+1 \\
& \texttt{MOV (}\mathcal{M}[\![\texttt{IBDISP (IMOV (BDISP r}_{13}\texttt{ d}_{13}\texttt{) (IREG NIL)) d}_5]\!]\ n+1) \\
& \quad (\mathcal{M}[\![\texttt{IREGDEF (IMOV (IMM v}_{10}\texttt{) (IREG NIL))}]\!]\ n) \\
= \;& \mathcal{T}[\![\texttt{IMOV (IMM v}_{10}\texttt{) (IREG NIL)}]\!]\ n \\
& \mathcal{T}[\![\texttt{IMOV (BDISP r}_{13}\texttt{ d}_{13}\texttt{) (IREG NIL)}]\!]\ n+1 \\
& \texttt{MOV (BDISP } n+1\texttt{ d}_5\texttt{) (REGDEF } n\texttt{)} \\
= \;& \mathcal{IR}[\![\texttt{IREG NIL}]\!]\ n \\
& \mathcal{IR}[\![\texttt{IMM v}_{10}]\!]\ n+1 \\
& \texttt{MOV (}\mathcal{M}[\![\texttt{IMM v}_{10}]\!]\ n+1\texttt{) (}\mathcal{M}[\![\texttt{IREG NIL}]\!]\ n\texttt{)} \\
& \mathcal{IR}[\![\texttt{IREG NIL}]\!]\ n+1 \\
& \mathcal{IR}[\![\texttt{BDISP r}_{13}\texttt{ d}_{13}]\!]\ n+2 \\
& \texttt{MOV (}\mathcal{M}[\![\texttt{BDISP r}_{13}\texttt{ d}_{13}]\!]\ n+2\texttt{) (}\mathcal{M}[\![\texttt{IREG NIL}]\!]\ n+1\texttt{)} \\
& \texttt{MOV (BDISP } n+1\texttt{ d}_5\texttt{) (REGDEF } n\texttt{)} \\
= \;& \mathcal{T}[\![\texttt{NIL}]\!]\ n \\
& \texttt{MOV (IMM v}_{10}\texttt{) (REG } n\texttt{)} \\
& \mathcal{T}[\![\texttt{NIL}]\!]\ n+1 \\
& \texttt{MOV (BDISP r}_{13}\texttt{ d}_{13}\texttt{) (REG } n+1\texttt{)} \\
& \texttt{MOV (BDISP } n+1\texttt{ d}_5\texttt{) (REGDEF } n\texttt{)} \\
= \;& \texttt{MOV (IMM v}_{10}\texttt{) (REG } n\texttt{)} \\
& \texttt{MOV (BDISP r}_{13}\texttt{ d}_{13}\texttt{) (REG } n+1\texttt{)} \\
& \texttt{MOV (BDISP } n+1\texttt{ d}_5\texttt{) (REGDEF } n\texttt{)}
\end{aligned}
$$

Figure 10: Transforming the tree code from figure 9 to linear code

```
reg(GBASE+r) = global(r).
reg(cwp+r)   = local(r) CWP[cwp].

global(r) = r =[ 0 ≤ r ≤ 9 ]⇛ .
local(r)  = r=[ 10 ≤ r ≤ 31 ]⇛ .

rs(0)   = reg(GBASE).
rs(res) = reg(dst) DST[dst] RES[res] =[ dst ≠ GBASE ]⇛ .
rs(val) = reg(ea) DST[dst] ea[val] =[ ea ≠ GBASE ∧ ea ≠ dst ]⇛ .

short(val) = REG rs(val).
           = IMM val.

scc(_)      = OFF.
scc(flags) = ON⟹CC[flags].

aflg(v) = (v <s) (v = 0) ¬(MINs ≤s v ≤s MAXs) (v >u MAXu) |.

arith(x,y,x+y)           = ADD.
arith(x,y,x+y+c)         = ADDC C[c].
arith(x,y,x−y)           = SUB.
arith(x,y,x−y−(1−c))     = SUBC C[c].
arith(x,y,y−x)           = SUBI.
arith(x,y,y−x−(1−c))     = SUBCI C[c].

arith(x,y,r) scc(aflg(r)) reg(ea) rs(x) short(y)
      =[ ea = GBASE ]⇛ .
  "   =[ ea ≠ GBASE ]⇛ ea[r].

PC[pc] pc[instruction]⟹IR[instruction].
IR[arith(x,y,r) scc(aflg(r)) reg(dst) rs(x) short(y)] PC[pc]
    ⟹RES[r] DST[dst] PC[pc+WORD].
DST[dst] RES[v]=[ dst = GBASE ]⇛ .
      "       =[ dst ≠ GBASE ]⇛ dst[v].
```

The number(s) between brackets indicate(s) the page(s) on which the concepts are defined. A reference to another concept in the glossary is indicated by a bold font.

access (44)

A synonym for **locator expression**.

alternatives (16)

A series of **update schemes**. The first **applicable** scheme in the series is applied.

applicable (4,21)

An **update scheme** is applicable if its left hand side is consistent with the current **configuration**, and its **guard** evaluates to true.

application phase (101,108)

In an implementation, the phase in which an update scheme is applied.

archetype (55)

A macro like mechanism.

archetype expansion (60)

The mechanism by which **archetypes** are expanded to give **update schemes**. Also the **text** parts of an **archetype** body.

archetype grammar (65)

A context free grammar representing the structure of the **archetype** declarations.

archetype, left handed (59)

An **archetype** having an empty right hand side **expansion**.

archetype, right handed (59)

An **archetype** having an empty left hand side **expansion**.

archetype, ambidextrous (59)

A pair of **archetypes** of the same name, one **left handed**, the other **right handed**.

backtrack rule (111)

A convention for indicating which **cells** need not be restored on backtracking.

box diagrams (49)

A diagrammatical notation for existential dependencies.

canonical form (96)

A desugared **update scheme** in which all **guards** have been made explicit.

casting (40)

Forcing the value of a term to be of a different type than the type globally declared for that term.

cell (3,19)

An element of a **configuration** or **memory**.

command driven (14)

An **update plan** in which all **update schemes** are **commands**

command form (14)

A **configuration** containing a non-empty **command sequence**, or one in which the contents of the **register** `PC` are not specified.

command sequence (14)

A sequence of terms.

command (6,12,14)

An **update scheme** in which both the left and right hand side are in **command form**.

completed left hand sides (109)

An **update scheme** in which the left hand side is extended to cover all **cells** covered by the right hand side.

configuration (19)

A partial function from **locators** to values.

consistent (3,20)

A **configuration** is consistent if it does not specify conflicting contents for one and the same **cell**.

context (60)

The non-**text** part of a configuration, in particular in an **archetype** body.

converse (109)

The **update scheme** obtained by swapping the left and right hand sides of a given **update scheme**.

correspond (110)

Two **locator expressions** correspond if they have the same left and right **locators**.

creation phase (100)

In an implementation, the phase in which internal representations of an **update scheme** are created.

dedicated register (44)

A **register** is dedicated to a **store** if it only appears as a **locator** of an **access** of that store.

derivation (4)

The result of **applying** an **update plan** to a **configuration**.

development (23)

The result of repeated **applications** of an **update plan** to a **configuration**.

direct access (44)

An **access** with a **locator** in a **register**.

equivalence (47)

See **semantic equivalence**.

expansion (60)

See **archetype expansion**.

expansion phase (105,106)

In an implementation, the phase in which **archetype** calls are (textually) **expanded**.

final configuration (4)

A **configuration** to which none of the **update schemes** in an **update plan** are **applicable**.

final development (4,23)

A **development** which is a **final configuration**.

finite semi-constant (9)

A term representing a finite set of constants.

fully structured store (44)

A **store** that is both **read structured** and **write structured**, and in which **read accesses** and **write accesses** share the same **register**.

ground term (42)

A term for which a variable free value can be derived.

guard (3)

A condition for **applicability**.

initial configuration (4)

Specifies the initial state of the memory before any **update schemes** are applied.

lee (45)

Cells that have already been passed by the **dedicated register** of a **structured store**.

locator (3,19)

An index to a **memory**.

locator expression (3)

A sequence of **cells**, delimited on the left and right by a **locator**.

left locator (10)

The **locator** on the left of a **locator expression**.

luff (45)

Cells yet to be passed by the **dedicated register** of a **structured store**.

memory (19)

A function from **locators** to values.

opcode (14)

The first element of a **command sequence**, if this is a constant.

outside (49)

A **semantic equivalence** or **translation** between two **plans** holds outside a set of **locators** if the property holds **within** the complement of that set.

parallel block symbol (open) (87)

'(||': indicates the start of a **parallel block**.

parallel block symbol (close) (87)

'||)': indicates the end of a **parallel block**.

parallel block (87)

A set of **update schemes**. All **applicable update schemes** are applied simultaneously. If the resulting right hand side is not **consistent** none of the **schemes** are applied.

parameter resolution (60,62)

The mechanism by which parameters of an **archetype** are rewritten to evaluable expressions.

pointer (44)

The contents of **dedicated register**.

probation phase (101,107)

In an implementation, the phase in which the **guard** is checked.

programme counter (6)

The **register** `PC`.

programme state (46)

The **configuration** of a **programme store** and of the **register** `PC`.

programme store (46)

A **store** having the **programme counter** as **dedicated register**.

programme (46)

The contents of a **programme store**.

queue (45)

A **semi-structured store** which reads and writes in the same direction.

read access (44)

An **access** on the left hand side of an **update scheme**.

read dedicated (44)

A **dedicated register** which addresses only **read accesses**.

read structured store (44)

A **store** in which all **read accesses** are **structured**.

read to the left (44)

A **direct access** the **left locator** of which is invariant.

read to the right (44)

A **direct access** the **right locator** of which is invariant.

register (10)

A constant **locator**.

repeat (12)

'"': indicates a repeat of the previous left hand side.

resolution phase (106)

In an implementation, the phase in which values are derived for terms.

restoration phase (106,107)

In an implementation, the phase in which provisional instantiations are undone.

right locator (10)

The **locator** on the right of a **locator expression**.

satisfied (20)

A **configuration** is satisfied by any **configuration** of which it is a subset.

script (4)

An **initial configuration** and an **update plan**.

selection phase (106,108)

In an implementation, the phase in which a choice is made between proceeding to the **verification phase** or continuing with the **expansion phase**.

semantic equivalence (47)

A property holding between two **update plans**.

semantic irrelevance (50)

A property of **configurations** with respect to an **update plan**.

semi-constant (9)

A regular expression over constants.

semi-ground term (42)

A term for which a variable free value can be derived.

semi-structured store (45)

A store which is **read structured** and **write structured**, but not **fully structured**.

statically semi-ground (42)

A **semi-ground term** for which a variable free value can be derived without reference to the current **configuration**.

store structure (41)

A regular expression over store names.

store (3)

A two way countably infinite set of **cells**.

textual expansion (60)

The replacement text of an **archetype** call, before **parameter resolution**.

structured access (44)

An **access** exhibiting a certain structure.

structured store (44)

A **store** in which all **accesses** are **structured**, and in which all these **accesses** share a common **register**.

text (17)

A sequence of terms.

textual expansion (60)

The replacement of an **archetype** by its definition.

translation (47)

A property holding between two **update plans**.

type alias (40)

A type declaration.

type grammar (41)

A context free grammar based on the structure of types.

type primitive (40)

A type name not appearing as the left hand side of a **type alias**.

unprotected variable (110)

A term in a **converse** which is not (**semi-**) **ground**.

update plan (4)

A set of **update schemes**.

update rule (3)

An **update scheme** containing no variables.

update script (4)

An **update plan** and an **initial configuration**.

update scheme (3)

Consists of a left hand side, a right hand side (both **configurations**), and a **guard**.

verification phase (101,108)

In an implementation, the phase in which the **guard** is checked.

within (49)

A **semantic equivalence** or **translation** holds within a set of **cells** if the property holds when cells not in the set are ignored.

write access (44)

An **access** on the right hand side.

write dedicated (44)

A **dedicated register** which addresses only **write accesses**.

write structured store (44)

A **store** in which all **write accesses** are **structured**.

write to the left (44)

A **direct access** the **right locator** of which is invariant.

write to the right (44)

A **direct access** the **left locator** of which is invariant.

Index